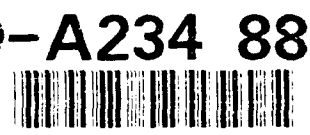

RADC-TR-90-404, Vol VI (of 18)
Final Technical Report
December 1990

BUILDING AN INTELLIGENT ASSISTANT: THE ACQUISITION, INTEGRATION, AND MAINTENANCE OF COMPLEX DISTRIBUTED TASKS

Northeast Artificial Intelligence Consortium (NAIC)

Victor R. Lesser and W. Bruce Croft

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This effort was funded partially by the Laboratory Director's fund.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-404, Volume VI (of 18) has been reviewed and is approved for publication.

APPROVED: Douglas A. White

DOUGLAS A. WHITE
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Ronald Raposo

RONALD RAPOSO
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Final Sep 84 - Dec 89

4. TITLE AND SUBTITLE
BUILDING AN INTELLIGENT ASSISTANT: THE ACQUISITION, INTEGRATION, AND MAINTENANCE OF COMPLEX DISTRIBUTED TASKS

5. FUNDING NUMBERS
C - F30602-85-C-0008
PE - 62702F
PR - 5581
TA - 27
WU - 13
(See reverse)

6. AUTHOR(S)
Victor R. Lesser and W. Bruce Croft

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeast Artificial Intelligence Consortium (NAIC)
Science & Technology Center, Rm 2-296
111 College Place, Syracuse University
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-404, Vol VI (of 18)

11. SUPPLEMENTARY NOTES (See reverse)
RADC Project Engineer: Douglas A. White/COES/(315) 330-3564

This effort was funded partially by the Laboratory Director's fund.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose was to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress during the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photointerpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is the development of intelligent interfaces to support cooperating computer users in their interactions with a computer.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Artificial Intelligence, Planning, Intelligent Computer-Aided Instruction, Intelligent Interfaces, Plan Recognition	128
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Block 5 (Cont'd) Funding Numbers

PE - 62702F	PE - 61102F	PE - 61102F	PE - 33126F	PE - 61101F
PR - 5581	PR - 2304	PR - 2304	PR - 2155	PR - LDFP
TA - 27	TA - J5	TA - J5	TA - 02	TA - 27
WU - 23	WU - 01	WU - 15	WU - 10	WU - 01

Block 11 (Cont'd)

This effort was performed as a subcontract by the University of Massachusetts at Amherst to Syracuse University, Office of Sponsored Programs.

Volume 6

1989 ANNUAL REPORT
To Rome Air Development Center

Building an Intelligent Assistant:
The Acquisition, Integration, and Maintenance
of Complex Distributed Tasks

Victor R. Lesser
W. Bruce Croft
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts

Contents

Section 6.1

Executive Summary

Our research over the five-year period has focused on two basic and interrelated research questions: 1) how to automate the acquisition, integration, and maintenance of a global understanding of a complex process from multiple, distributed local perspectives, 2) how to use this "understanding" to support users in their cooperative interaction with the system and other users by assisting in the execution of tasks and by explaining the reasons behind the actions and decisions involved in reaching the current state of the system. We believe these are some of the key research questions that need to be solved in order to effectively use a distributed network of workstations to perform cooperative problem solving in a complex task-oriented environment. The importance of this problem area and related research issues has been increasingly recognized over the five years of this research project. We think that the results of our research will, and to some degree already have had, a significant impact on the emerging subfields of intelligent interfaces and computer-supported cooperative work.

During the five years of the project, we have studied these issues by looking at the task domains of software development, office procedures, project management, and tutoring. We have built a number of systems to demonstrate our research. Our early work in an intelligent assistant focused on understanding user actions through hierarchical event-based descriptions. An offshoot of this work was the approach of using event-based behavioral abstractions as a tool for distributed debugging and performance evaluation. This approach is now receiving wide attention. As a result of this early work, we realized that advances in the following basic research areas would be necessary for satisfying the goals of our project. For each of these areas, we list our research accomplishments:

- Knowledge Representation — The goal is to provide a framework for representing realistic models of complex open-ended domains. The results were:
  - a knowledge representation framework that integrates activity models, agent descriptions, object specifications and relationships;
  - meta-plans as a specification technique for large and complex plan libraries and domain-dependent exception handling routines;
  - integration of empirical knowledge which represents soft domain constraints into classic hierarchical plan formalisms through the addition of a truth maintenance system (TMS);

- integration of planning and simulation techniques for validating plans in complex, dynamic environments.

- Knowledge Acquisition — The goal is to develop techniques for user specification of plans and dynamic acquisition at run time of new and revised plans. The results were:
  - an approach for knowledge acquisition based on the assimilation of new and revised plans with existing specifications;
  - a cognitive model for how people recall their activities and an interface based on this model that can be used to acquire plans, display plans and modify plans;
  - a formal model of plan exceptions, and techniques for detecting, classifying and learning from them.

- Focusing in Plan Recognition — The goal is to develop techniques for quickly and efficiently arriving at the best interpretation of the actions/data that have currently been observed. The results were:
  - a recognition architecture that exploits heuristics based on user rationality and fully uses available constraints as soon as possible in the recognition process;
  - a new approach to controlling plan recognition, called evidence-based plan recognition, that uses a symbolic representation of current uncertainties in the interpretation and control plans keyed to specific uncertainties to efficiently guide the recognition process.

- Multi-Agent, Interactive Planning — The goal is to provide a framework to specify partial plans and for the user(s) to interact with planner(s) to complete these plans based on the dynamics of the specific situation. The results were:
  - an architecture based on a formal model of interactive planning has been implemented;
  - models for negotiation among user and system, and among systems, have been developed.

- Knowledge Display — The goal is to provide a display framework and tools for the user to effectively interact with an intelligent assistant:
  - a suite of programming tools that enables authors to browse and explain knowledge in an expert system for tutoring. These tools facilitate tracing and summarizing the reasoning within an expert system and allow an author to interactively modify system reasoning and response in an intelligent discourse system;
  - a graphics object-oriented environment for building simulations of complex environments for decision support.

These ideas have been realized in a number of systems. The GRAPPLE system monitors a user's activities, detects errors, and reasons about the user's plans. It uses domain knowledge to make plausible assumptions about missing values in an open world application. This is used to provide improved error detection, prediction, and disambiguation.

POLYMER is a planning system which constructs partial plans and executes them interactively. It uses constraints from agent actions to extend its partial plans. Exception handling

is achieved by a subsystem called SPANDEX, that classifies exceptions and constructs an explanation of how each action may fit into the current plan. Plan acquisition is supported by a subsystem called DACRON, that has a graphical interface based on how people recall their tasks, and a knowledge assimilation is supported by a subsystem called KnAc.

In short, we feel that significant progress, both from a conceptual and practical perspective, has been made in developing an intelligent assistant to support knowledge-based, computer-supported cooperative systems. Additionally, we think there has been important spin-off research in planning and plan recognition that will also have significant impact.

We have also built a suite of software tools that enable a user to browse through a complex knowledge base while the system tailors its explanation to the particular user. Other tools facilitate tracing and summarizing expert system reasoning.

Both the GRAPPLE and POLYMER systems have been built during the past 5 years. Each recognizes user plans and monitors user activities. It handles exceptions to given plans. GRAPPLE makes plausible assumptions about missing values in an open world application and POLYMER constructs partial plans and employs constraints based on user activities to extend these partial plans.

Several systems are designed to improve the quality of an intelligent interface. We have developed several knowledge acquisition systems, one for the acquisition of user plans and another for the acquisition of discourse decisions. The latter systems enables a user to input new choices about machine interventions and responses based on its internal user model and discourse history. This system enables an author to expand the system's repertoire of response even as testing and evaluation of the system proceeds.

Section 6.2

Ancillary Activities

6.2.1 Degrees Conferred

- Ph.D.s: 1
- MSs: 2

6.2.2 New AI Faculty

- 1. Professor Rod Grupen

6.2.3 New AI Courses

- 591D (Fall '89): Computer Vision
- 591L (Fall '88): Image Processing
- 689 (Spring '89): Machine Learning
- 691C (Spring '89): Topics in Optimization
- 791A (Fall '88): Connectionist-Approach Learning
- 791A (Fall '89): Mobile Robots
- 791W (Fall '88): Computer-Supported Cooperative Work
- 791W (Spring '89): Knowledge-Based Tutoring and Advisory Systems
- 791X (Fall '88): Sophisticated Control of Knowledge-Based Systems

6.2.4 Journals, Book Chapters, Other Papers

Broverman, C., "Plan Execution Using Human Agents," Department of Computer and Information Science, University of Massachusetts at Amherst, Technical Report 89-83.

Carver, N. and Lesser, V.R. "Planning for the Control of an Interpretation System," University of Massachusetts/Amherst Computer and Information Science Department Technical Report 89–39, April 1989.

Connell, M. Huff, K.E. and Lesser, V.R. "Implementing an Incremental Hierarchical Plan Recognition System," *Proceedings of the 2nd AAAI Workshop on Plan Recognition*, IJCAI-89, Detroit, 1989.

Huff, K.E. and Lesser, V.R. "A Plan-Based Intelligent Assistant that Supports the Software Development Process," *Third ACM Symposium on Software Development Environments*, Boston, Massachusetts, November 1988.

Kuwabara, K. and Lesser, V.R. "Extended Protocol for Multistage Negotiation," *Proceedings of the 9th Distributed Artificial Intelligence Workshop*, 1989, pp. 129–161.

Lander, S. and Lesser, V.R. "A Framework for the Integration of Cooperative Knowledge-based Systems," *Proceedings of the 4th IEEE International Symposium on Intelligent Control*, Albany, NY, September 1989, pp. 472–477.

Lander, S. and Lesser, V.R. "A Framework for Cooperative Problem-solving Among Knowledge-based Systems," *IJCAI 1989 Workshop on Integrated Architectures for Manufacturing Working Notes*, Detroit, August 1989.

Lander, S., "A Framework for the Integration of Cooperative Knowledge-Based Systems," submitted to the Fourth IEEE Symposium on Intelligent Control. 7/89

Lefkowitz, L.S. and Lesser, V.R. "Knowledge Acquisition as Knowledge Assimilation," *International Journal of Man-Machine Studies*, Volume 29, No. 2, 1989, pp. 215–226.

Mahling, D. and Croft, W.B. "Relating Human Knowledge of Tasks to the Requirements of Plan Libraries," *International Journal of Man-Machine Studies*, 31, 1989, pp. 61–97.

Mahling, D. and Croft, W.B. "Knowledge Acquisition for Planners," *Knowledge Acquisition*, (to appear).

Croft, W.B. and Lefkowitz, L. "Knowledge-based Support for Cooperative Activities," *Proceedings of HICCS-21*, 1988, pp. 312–318. (Also in *Readings on Distributed AI*, Morgan Kaufmann, 1988.)

Lefkowitz, L. and Croft, W.B. "Planning and Execution of Tasks in Cooperative Work Environments," *Proceedings of the Fifth IEEE Conference on Artificial Intelligence Applications*, Miami, Florida, 1989, pp. 255–262.

Mahling, D. and Croft, W.B. "A Visual Language for the Acquisition and Display of Plans," *Proceedings of IEEE Workshop on Visual Programming Languages*, 1989, pp. 50–56.

Woolf, B., "Representing, Acquiring and Reasoning about Tutoring Knowledge," *Proceedings of the Second Planning Workshop for Intelligent Tutoring Systems* AFHRL, Brooks Air Force, San Antonio, TX. (chapter)

Woolf, B., "Tutoring Environments for the Construction and Communication of Knowledge." chapter in book entitled Design Principles for Building Software for Learning, 1989.

Woolf, B., "Computer Partners in our Future," co-authored by Ted Slovin, published in "The Futurist: A Journal of Forecasts, Trends and Ideas about the Future."

6.2.5 Progress or Products Resulting from NAIC Research

TECHNOLOGY TRANSFER:

Beverly Woolf gave presentations at UMass for visitors from external research at Apple Computer Company and the Electricity Commission of Victoria, Australia.

Beverly Woolf presented her work to External Research, Apple Computer Company in Cupertino, CA July 18, 1989 and to the Multimedia Lab, Apple Computer in San Francisco July 19, 1989.

SEMINARS/WORKSHOPS:

Scott Anderson and Penni Sibun attended the Meeting of the Association of Computational Linguistics in Seattle June 26–30, 1989.

Scott Anderson attended the 2nd Annual CUNY Conference on Human Sentence Processing.

TECHNICAL PRESENTATIONS:

David Lewis delivered talks at Cornell, Carnegie-Mellon and Unisys to discuss the use of Natural Language Processing for Information Retrieval with researchers.

David Lewis gave a talk on the CL model and representation quality in Information Retrieval as part of the Office of Naval Research, URI, site visit, July, 1989.

Bev Woolf presented her work at an NSF Principal Investigators meeting in Pittsburgh on November 3 and 4, 1988 and was an invited speaker at a workshop on Hypermedia, sponsored by General Electric, Schenectady, New York, November 28, 1988.

Beverly Woolf described her work to a gathering of University professors and Apple employees at a meeting of Apple External Research, Cupertino, CA, January 23, 1989.

Beverly Woolf presented a seminar of her work for Andy Van Dam at Brown University March 1, 1989 and was an invited panelist at the American Educational Researchers Association in San Francisco, March 27, 1989.

Beverly Woolf delivered a talk entitled "Representing, Acquiring and Reasoning about Tutoring Knowledge," at the Intelligent Tutoring Systems Research Planning Forum at the Air Force Human Resources Laboratory, Brooks Air Field, San Antonio, Texas, April 5–6, 1989.

Beverly Woolf gave invited presentations at the International Conference on Computer Assisted Learning, University of Texas at Dallas, May 10 and at the University of Maine at Orono May 23, 1989.

Beverly Woolf delivered a seminar at NASA training headquarters in Houston, Texas and presented a paper at an AI Workshop for American Express Executives, in Minneapolis, MN, May 12, 1989.

Beverly Woolf presented a demo and a talk describing her NLP/planning system at the Rome Air Force Development Center, July 27, 1989.

6.2.6 Improvements to Research Environment

During this last year we have significantly upgraded our computational capabilities in two ways; we've increased the capabilities of existing Lisp machines, and have added new machines (see list below). As a result of these additions, we have over sixty Lisp machines in our environment.

12 TI Explorer-I to Explorer-II upgrades
2 TI Explorer-I to Explorer-II-plus upgrades
1 TI Explorer-II to Explorer-II color upgrades
7 TI Micro-Explorers
2 TI networked micro-Explorer Systems
2 Sun 4/110FCE
1 MacIvory

Section 6.3

Overview of Research

6.3.1 *Planning and Plan Recognition*

6.3.1.1 Deeper Domain Modeling

We show that intelligent assistance for software development is an *open world*, where useful information about the state of the world is missing. This proves to be a significant barrier to achieving a nontrivial representation of the domain and leaves the intelligent assistant without a basis for independently critiquing many actions of the user. We advance a solution to acquiring additional state information by using domain knowledge to make plausible assumptions about the missing values based on the observed state. This process is formalized as non-monotonic reasoning. Using these plausible assumptions, the *credibility* of competing alternatives can be evaluated independently of the user; actions that are consistent with current assumptions will have the highest credibility. If it becomes necessary, actions that have low credibility can still be pursued after *reconciling* the assumptions with the requirements of the actions. The ability to make plausible assumptions allows additional domain knowledge about the context for actions to be made explicit; thus the approach enables deeper modeling of the domain.

6.3.1.1.1 Approach to Deeper Domain Modeling

Substantive support of processes requires involvement in both the complex decisions as well as the mundane details. We will show that formally representing the knowledge involved in some of these decisions can be a challenge. As examples in the software development application, consider the criteria for choosing the baseline from which to develop a new system version, selecting tests to run, or deciding which system version is releasable. If a process assistant lacks knowledge to address these decisions, it cannot independently critique a choice made by the user, nor can it suggest a restricted set of likely candidates from which the user can choose. In this case, intelligent assistance is seriously restricted because the representation of the domain is limited to surface issues.

6.3.1.1.2 The Open Worlds Problem

Consider the goal of testing a new system version. To test a system means running all applicable testcases, and only those cases (neither under- nor over-testing is desirable). The problem is that

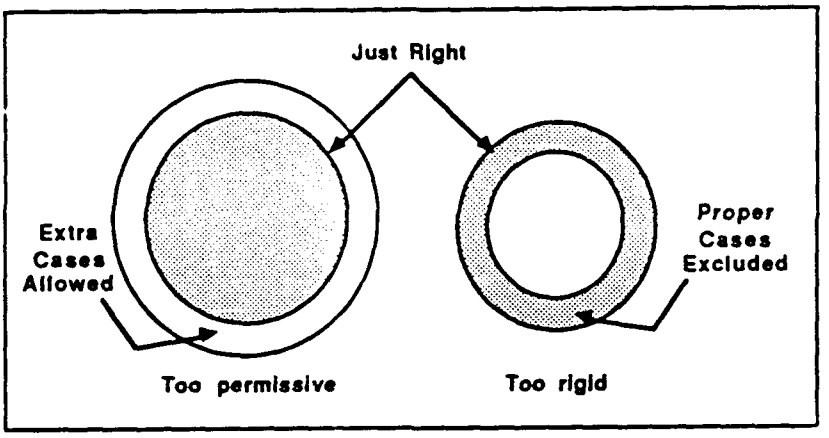

Figure 1: Rigidity versus Permissiveness in Plan Recognition

applicability of testcases cannot be directly observed as a result of past actions nor computed with certainty from observable data. Thus, when a testcase is run, there is no independent basis for determining if the testcase is indeed applicable, or if this is the last of the applicable test cases (which signals the end of testing). Knowledge about testing strategies (where a given strategy implies that certain categories of testcases are applicable) underlies the choice of testcases, but this deeper knowledge cannot be exploited since the operative testing strategy cannot be determined—it too is not observable.

When information about the state of the world is incomplete, we say that a planning application involves an *open world*. The software development application is an open world since predicates indicating whether a testcase is applicable or whether a system version is releasable are not directly observable, nor are they readily computable from the observable data. Open worlds present significant problems. When it is impossible (or infeasible) to acquire the missing information, plan recognition is adversely affected. Competing interpretations of actions cannot be disambiguated, some distinctions between legal and illegal actions/plans cannot be made, predictions of future actions lack precision, and there is no basis to explain why observed actions/plans were chosen over other alternatives.

These problems with planning in an open world are due to the difficulty of achieving accurate descriptions of operators. Predicates whose truth or falsity cannot be determined cannot be used in operator definitions to define the relevancy of an operator (preconditions), the decomposition and completion criteria (subgoals), or restrictions on parameter bindings (constraints). When such predicates are omitted entirely, the operator definitions are under-constrained and plan recognition is too permissive; N "illegal" plans will be accepted, predictions will be too general, and alternatives that are actually irrelevant cannot be discarded. Attempting to compensate by substituting another expression for the missing predicate may yield a definition that is over-constrained and plan recognition will be too rigid.

In intelligent assistance, as in other applications of plan recognition to open worlds, it is important to find some balance between these two extremes of permissiveness and rigidity (as illustrated in Figure 1). An approach is needed that will closely approximate the desired situation, without being consistently too permissive or consistently too rigid. Further, the approximation should be "elastic," in the sense that it can be adjusted when a deviation is identified. This is

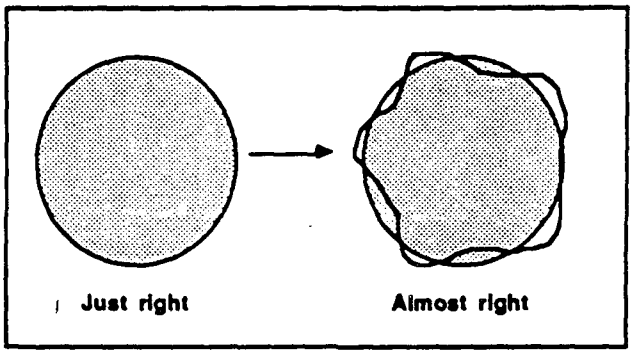

Figure 2: Finding a Balance between Rigidity and Permissiveness

shown in Figure 2.

6.3.1.1.3 Towards a Solution

One approach that captures the spirit of Figure 2 is to use the observable state information to make plausible assumptions about the missing state information. In the testing example, there is a correlation between the types of changes made during source editing (something that can be measured) and the operative testing strategy, which in turn determines the categories of tests that are applicable. For example, when changes are simple (affecting a few lines of code), a weak test strategy would typically be appropriate—only base testcases would be applicable. Otherwise, standard testing would typically be appropriate—base and normal testcases would be applicable. This reasoning is inherently non-monotonic. Assumptions are made in the absence of information to the contrary; later, additional information may be acquired that defeats the earlier conclusion and its consequences.

When assumptions about test strategy and applicability are added to the observable state of the world, it is possible to evaluate the *credibility* of alternatives. Interpretations of actions (or predictions of future actions) that agree with the current assumptions have the highest credibility; interpretations lose credibility with each assumption that is violated. If the operative testing strategy is assumed to be standard testing, then an action to run one of the normal cases is fully credible, because that case is assumed to be applicable. On the other hand, an action to archive the system, which is predicated upon testing being completed, would have less than full credibility as long as there are applicable (i.e., base or normal) cases still to be run.

Given two alternative interpretations that differ in credibility, the more credible alternative is more likely to be the correct interpretation. Given two choices for completing an unsatisfied subgoal, the more credible alternative is the better prediction of the future. An action whose "best" interpretation is below a certain credibility threshold is a possible user error. Thus, the programmer can be advised of a possible oversight when archiving prior to running all applicable testcases. Finally, it is possible to give the underlying reason for the credibility of running or not running a particular testcase, by citing the operative testing strategy and its implications.

Credibility can be combined with other discriminators to determine which interpretations

to pursue. Sometimes it will be necessary or desirable to pursue an interpretation that is not fully credible; for example, the interpretation may still be the "best" considering all available discrimination information. In order to proceed with such an interpretation, it is necessary to *reconcile* the assumptions about the state of the world with the requirements of the desired interpretation. For example, suppose the operative testing strategy is assumed to be standard testing. To pursue an interpretation in which archiving starts when only base cases have been run, it is necessary to revise the assumption that testing is not done. While this can trivially be accomplished by simply recording that testing is done, it is far more interesting to provide some rationale for testing being done. And, indeed, the preferred reconciliation is to change the operative testing strategy from standard to weak testing, after which it follows that testing is now done. (The full implementation of all the examples informally described here will be given later).

In the remainder of this section, we show how a plan recognition system, based on the classical hierarchical planning paradigm, can be extended to incorporate a new type of domain knowledge about the context for actions; unlike the knowledge already reflected in operator definitions, this knowledge is approximate rather than absolute. This approach allows plausible inference and plausible explanation within a deeper model of domain activities than is otherwise possible. In section 6.3.1.1.4, we show how to capture this deeper knowledge, and how to exploit it for reasoning about world state. This is accomplished through monotonic and non-monotonic rules in a truth maintenance system. In section 6.3.1.1.8, we explore the impact on a plan recognition architecture. *Credibility* represents a new perspective from which competing interpretations can be disambiguated. *Reconciliation* is the means by which assumptions are revised when the plan recognizer discovers that its assumptions are wrong.

6.3.1.1.4 Reasoning About the State of the World

Filling in the missing details about the current state of the world is a process of extrapolation from what is known about the state. Extrapolation involves adding certain specific conclusions about the state when a particular pattern of other propositions holds in the state. Thus, extrapolation lends itself naturally to a rule-based process: *when* < *pattern* >, *add* < *conclusion* >. In this approach, the additional knowledge that enables deeper domain modeling is captured in these rules, not directly in the operators (which are the standard vehicles for expressing domain knowledge). Thus, actions determine a *core* state; the rules are then applied to the core state to arrive at an *extended* state. If the additional knowledge were to be expressed directly in the operators, individual rules would have to be replicated in multiple operators. (A pattern can be an arbitrarily complex logical expression involving predicates whose truth is determined by many different operators; each such operator would have to use the pattern in a conditional effect).

It is a requirement that the rule system support non-monotonic reasoning, in order to capture the conjectural nature of some of the conclusions. Non-monotonic reasoning allows a connection between pattern p and conclusion c that is a special kind of logical implication; this connection between p and c is such that p *typically* implies c (that is, *if there is no information to the contrary*, then c holds if p holds). Different systems for non-monotonic reasoning, including circumscription, modal logics, and default logic, take different approaches to formalizing the concept of "no information to the contrary." For an overview of non-monotonic reasoning, see [36].

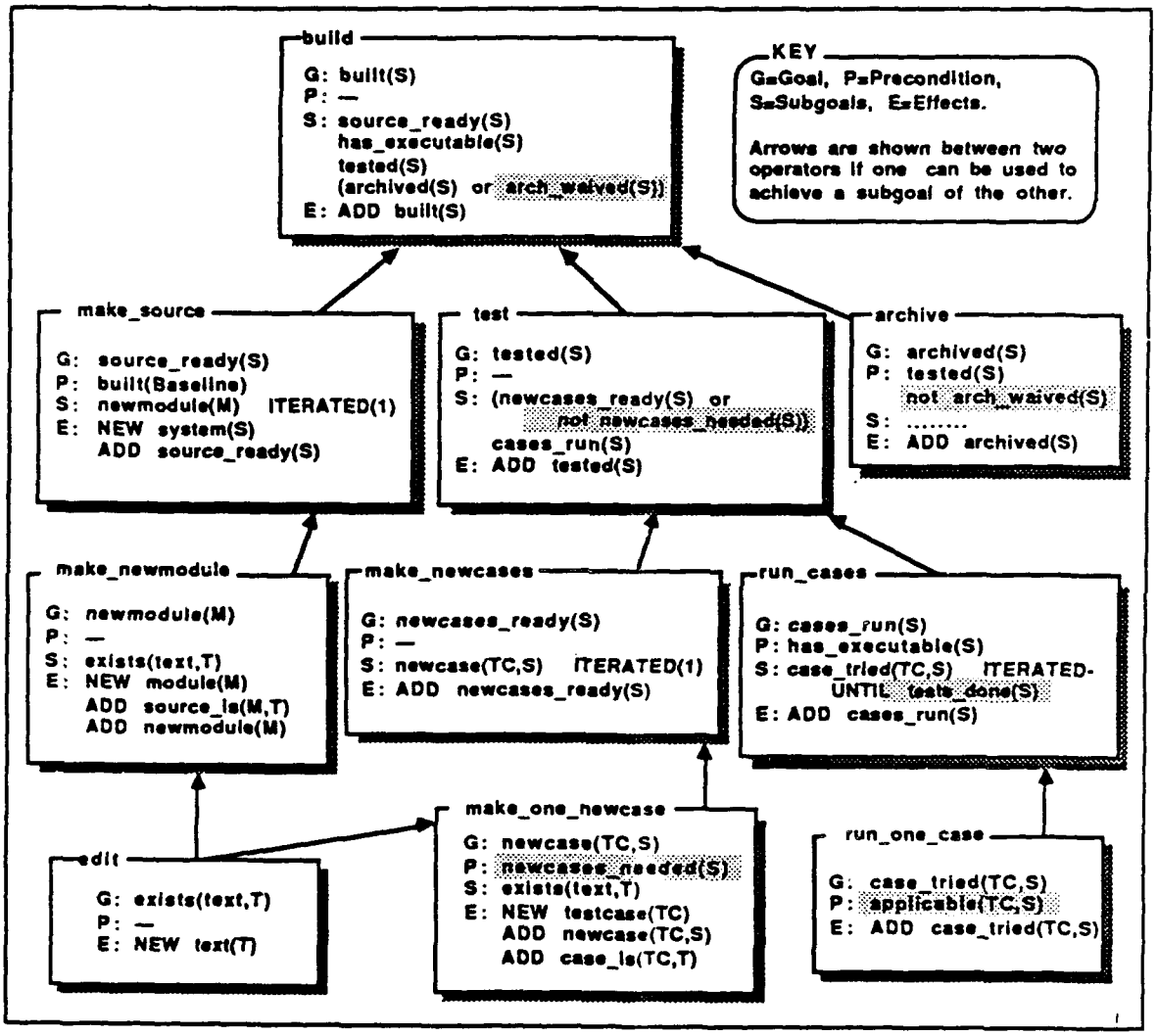

Figure 3: Operators for Building a System Version

A truth maintenance system (TMS) [28] is one approach to implementing non-monotonic reasoning, based on multi-valued logic. (Although a similar concept, the assumption-based truth maintenance system, ATMS [deKleer, 1986], has been introduced, we are interested here in the original notion of a TMS as described by Doyle.) A TMS maintains a network of nodes, each of which can be labeled IN or OUT. Separate nodes are used for a proposition and its negation. (A proposition is a predicate with bound arguments; if the predicate is a core state predicate, then the proposition is a core state proposition). If the node for a proposition is IN and the node for its negation is OUT, the proposition is true; if the node is OUT and the negation is IN, the proposition is false. If both are OUT, the truth value is unknown; if both are IN, there is a contradiction.

Justifications capture the relationships between the nodes, correlating a set of support nodes and a set of exception nodes with a conclusion node. A justification of the form A EXCEPT B→C *means that if* A is IN and B is OUT then C is IN. The exception node B represents the non-monotonic content of the justification; a monotonic justification has an empty list of exceptions. In order for a node to be IN, it must have at least one valid justification; a justification is valid if all its support nodes are IN and all its exception nodes are OUT. A premise justification has empty support and exception lists, so it is always valid.

6.3.1.1.5 Domain Knowledge in TMS Justifications

As an example, take the selection of operators in Figure 3 for building a system version. The effects of these operators determine the core state of observable facts. The use of the extended state predicates that cannot be observed is highlighted. For example, the precondition in *run_one_case* requires that the testcase be *applicable*; the iteration completion criteria in *run_cases* is that *tests_done* be true; and, there is a subgoal in *build* that will allow archiving to be skipped when waiving it is appropriate. The objective is to use TMS justifications to derive truth values for these extended state predicates.

Example domain knowledge about *applicable* and *tests_done* is given in justification form in Figure 4. In order to express this knowledge, several additional predicates have been introduced. Changes made during editing are used to conjecture whether the test strategy should be standard or not (rules J1-J2). The non-monotonic rule J1 can be read "In the absence of information to the contrary, if substantive changes are made during editing, then a standard testing strategy is appropriate." Testing strategy determines whether normal testcases are relevant (rules J4-J5); base cases are always relevant (rule J3). Rules J6-J7 establish that cases that are relevant are applicable unless they are specifically excluded (as they would be if infected by some catastrophic bug that is not yet fixed). Rule J8 states that typically cases are not specifically excluded. Tests_done does not hold until all applicable testcases have been tried (J9-J10). Note that both monotonic and non-monotonic justifications have been used; the meaning of the testing strategies is defined monotonically, for example.

These justifications are predicated upon being able to determine the extent of changes made to the source code for a system during editing. The easiest way to do this is to measure the difference between the source of the baseline for the system and the source of the system, which can be done at the time of linking (when it is known for certain that all source editing is completed). That means that the link action should actually be a command language script that both calls the link editor and performs the difference operation. In this way, *substantive-changes* is a predicate

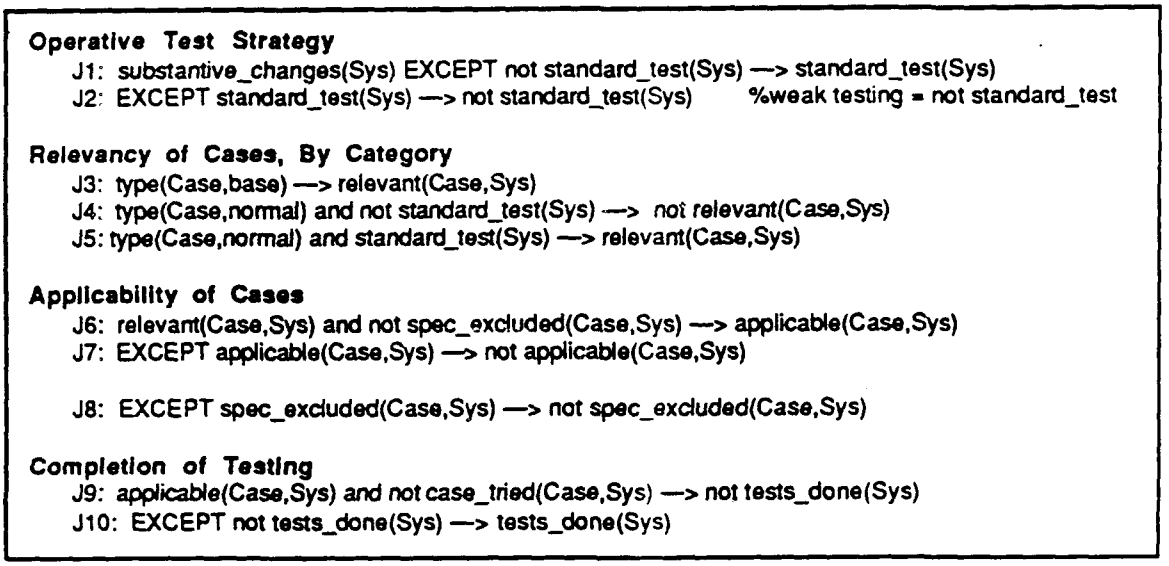

Figure 4: Example Justifications

that can be set by the effects of the *link* operator .

TMS justifications bear a strong resemblance to rules in default logic [69], an alternative, and more general, approach to non-monotonic reasoning. In default logic, a rule is written A:MB/C to mean that if A is provable and B is consistent (i.e., B is not provable), then C can be concluded. A TMS translation would be A EXCEPT B→C. This simple translation hides fundamental differences, and their practical implications. The TMS deals with validity (truth in a model), while default logic deals with provability (validity in all models); for the approach to open worlds described here, the TMS facilities are adequate. (The formal relationship between the two systems has yet to be studied in detail [32]). As an example of practical implications of these differences, consider a rule set in default logic and the equivalent rule set expressed in TMS form. Since an extension is the default logic concept corresponding to a TMS labeling, we might expect the extension and labeling to give the same answers. However, they do not always do so: for example, there are rule sets having multiple extensions and exactly one labeling [60].

6.3.1.1.6 Representing World State in the TMS

Justifications provide a way to derive the extended state from the core state. When the justifications are instantiated, and the truth values for core state propositions entered (with premise justifications), the truth maintenance process will label the nodes, giving a read-out on the truth values for the extended state propositions. When the core state changes as a result of an action, the premises will change, nodes will be relabeled, and the extended state propositions may change; in this way, the extended state propositions may vary over time. The TMS will also determine whether the truth value of a given proposition is *certain* or *by-assumption*. A proposition is certain unless one or more non-monotonic rules were used to determine its truth value. (The truth values of propositions that are certain cannot be changed in order to reconcile an

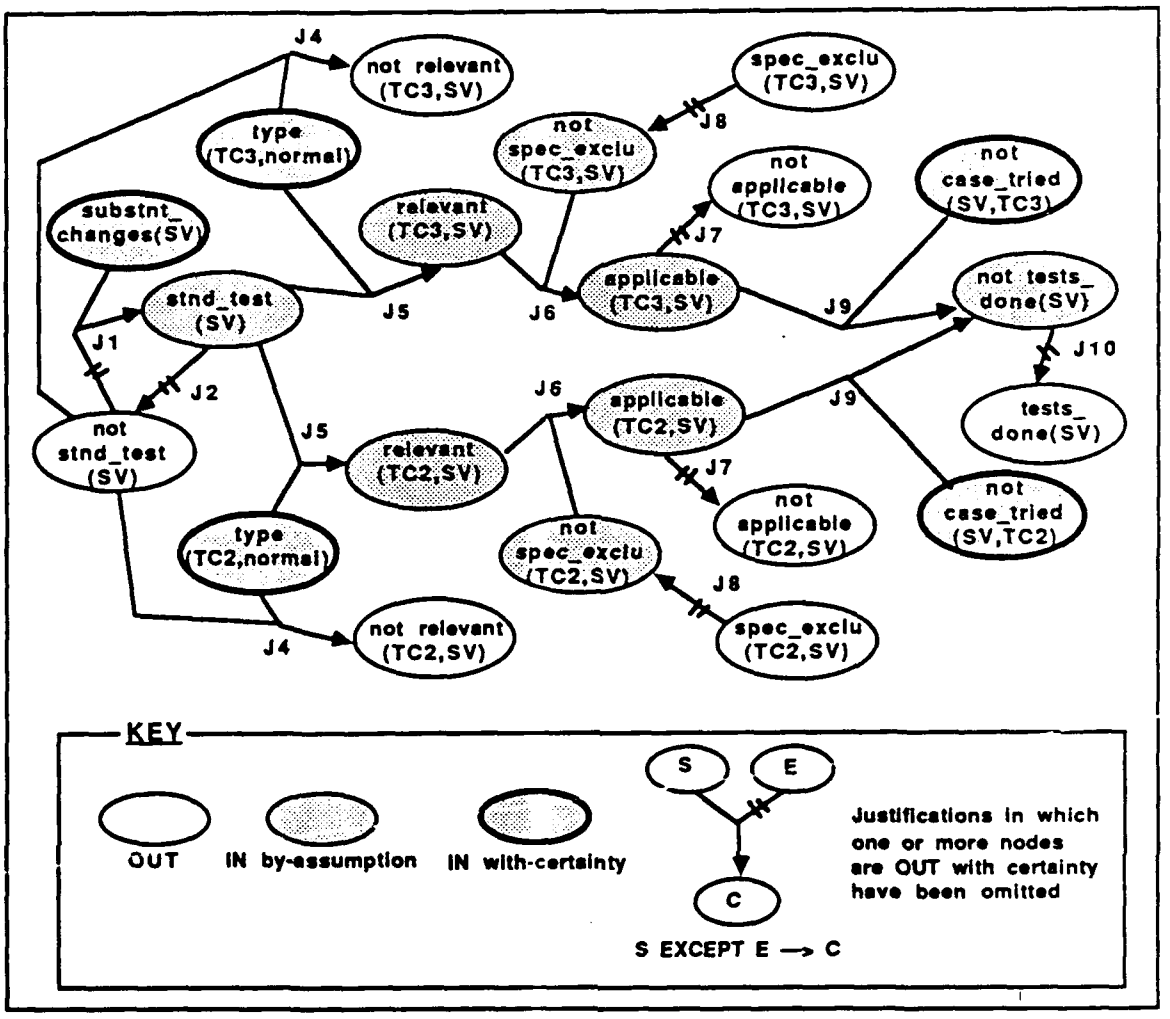

Figure 5: Instantiated Justifications

interpretation).

As an example, consider a situation where the building of system version SV has progressed to the point where testing is in progress; suppose also that the source editing changes were substantive. Let there be three testcases, where TC1 is a base case, and TC2 and TC3 are normal cases; suppose TC1 has been run. The state of instantiated justifications is given in Figure 5, showing that standard testing is assumed to be operative (conclusion of J1), that TC2 and TC3 are assumed to be applicable in testing SV (conclusions of two instances of J6), and that testing is not done as there are (two) applicable cases that have not yet been run (conclusions of two instances of J9).

It should be noted that this use of a TMS is somewhat different from traditional uses. In a traditional TMS application, the problem solver using the TMS is constantly discovering new justifications; this discovery process is guided by the current state of the TMS. At any given time, the current justifications represent only partial knowledge concerning the interrelationships

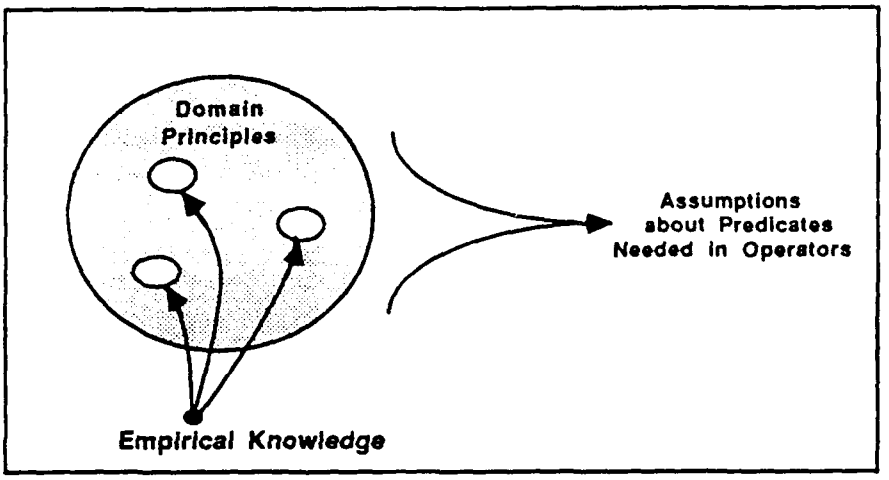

Figure 6: Knowledge in Justifications

among the facts about the world. And, the TMS is used to model successive approximations of the same state of the world, not successive states. Our use of the TMS differs in two ways. First, all justifications are predefined; there is no dynamic discovery of new interrelationships expressed in new justifications. Second, the TMS is used to model successive states of the world (shown clearly by the fact that premises can be retracted).

6.3.1.1.7 Implications for Deeper Domain Modeling

We have shown how justifications and the labeling mechanism of a TMS can be used to extrapolate from observable information to fill in missing facts about the state of the world that are needed to define domain operators. We need to characterize the nature of the domain knowledge that is captured in the justifications in order to understand why this approach allows deeper domain modeling. We do so by first discussing some of the characteristics of the example we have developed, and then generalizing.

The justifications shown in Figure 4 are all directed at supplying truth values for the predicates *applicable* and *testing_done*; these predicates are needed in the operators *run_one_case* and *run_cases* respectively (as shown in Figure 3). It turns out that neither of these predicates can be estimated directly from observable predicates. Rather, there is a whole chain of reasoning that leads to approximating the predicate *applicable*, and that predicate is then available to be used in the approximation of *testing_done*. This reasoning, embodied in justifications J3-J7 and J9-10, captures background knowledge about the programming process domain. In this case, the background knowledge involves such notions as testing strategies, and how the operative testing strategy affects test case selection.

In order to be able to apply this background knowledge, other predicates have to be approximated; without a way to evaluate *standard_test* or *spec_excluded*, the background knowledge cannot be exploited. The justifications that do this (J1-2 and J8) capture empirical relationships. Justification J1, for example, cites a correlation between substantive editing changes and the standard testing strategy, without explaining the correlation in any way.

From this description, we can see that the knowledge captured in the justifications is of two sorts: background knowledge capturing fundamental principles about the domain, and empirical

knowledge about observed relationships. As shown in Figure 6, the background knowledge by itself is incomplete, and the empirical knowledge is needed to fill the gaps. It is the background knowledge, not the empirical knowledge, that actually contributes to deeper domain modeling; if only empirical knowledge were involved, there would be no deeper understanding of the domain. The empirical knowledge is the source of the uncertainty in the reasoning. (In our example, the two justifications J7 and J10 are in non-monotonic form for convenience; the uncertainty in *applicable* and *tests_done* actually arises from justifications 1, 2 and 8).

Although the primary objective of capturing additional domain knowledge was to define selected predicates that were needed in operator definitions, the approach we have taken has actually resulted in the definition of a much larger set of new domain predicates, as well as the definition of relationships (theoretical and empirical) among these predicates. Many of the new predicates do not appear directly in operator definitions, but their truth status affects that of predicates in the operators. In this way, the justifications capture knowledge about the context for actions and allow plausible inference within this previously-inaccessible context.

6.3.1.1.8 Impact on Plan Recognition Architecture

In a conventional plan recognition algorithm, propositions describing the world state evaluate to true or false, the evaluation is known to be certain, and thus interpretations are either valid or invalid. With the introduction of TMS justifications, there is a middle ground between valid and not valid—described by degree of credibility. In this section, we discuss how a plan recognizer can capitalize on this new information, as well as deal with the fact that assumptions about the state of the world may be wrong.

Effective plan recognition involves making timely and knowledgeable choices between competing interpretations. Deferring decisions, thereby allowing further actions to narrow the field, restricts the ability to make inferences about the current situation. Making arbitrary choices is computationally expensive since they often have to be re-visited. In [49], the domain-independent assumption that the preferred interpretation contains a minimal number of top level plans is used to make reasoned choices in the presence of uncertainty. In [17], both domain-independent and domain-dependent heuristics could be expressed and used to guide choices. Credibility represents an additional source of discrimination knowledge, as described below.

6.3.1.1.9 Use of Credibility

As an example of the use of credibility, consider the following scenario. Let building of system version SV be in progress as described in Figure 5, where substantive changes were made during source editing, standard testing is assumed operative, cases TC1-TC3 are assumed applicable, and only case TC1 has been run. Further, let there be assumptions that new testcases are needed, and that archiving is not waived. The state of the plan for building SV is shown in Figure 7. Let the next action be *edit*. Given the operator library of Figure 3, there are two interpretations for *edit*: editing to make a new testcase and editing to get source code ready.

Consider the interpretation of *edit* as part of making new testcases; in this interpretation (Figure 8), *edit* continues the work of building system SV. *Edit* itself has no preconditions, but it inherits the precondition *newcases_needed* from *make_one_newcase*. Since *newcases_needed* is true by-assumption, this interpretation has high credibility—it depends on (one) extended state

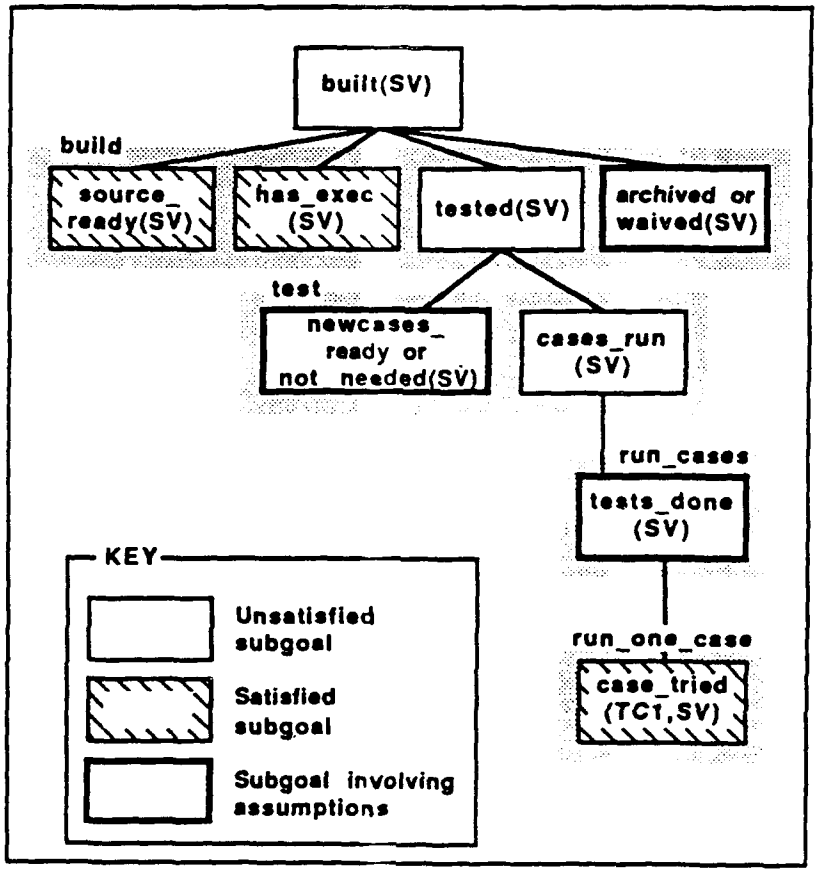

Figure 7: A Plan in Progress

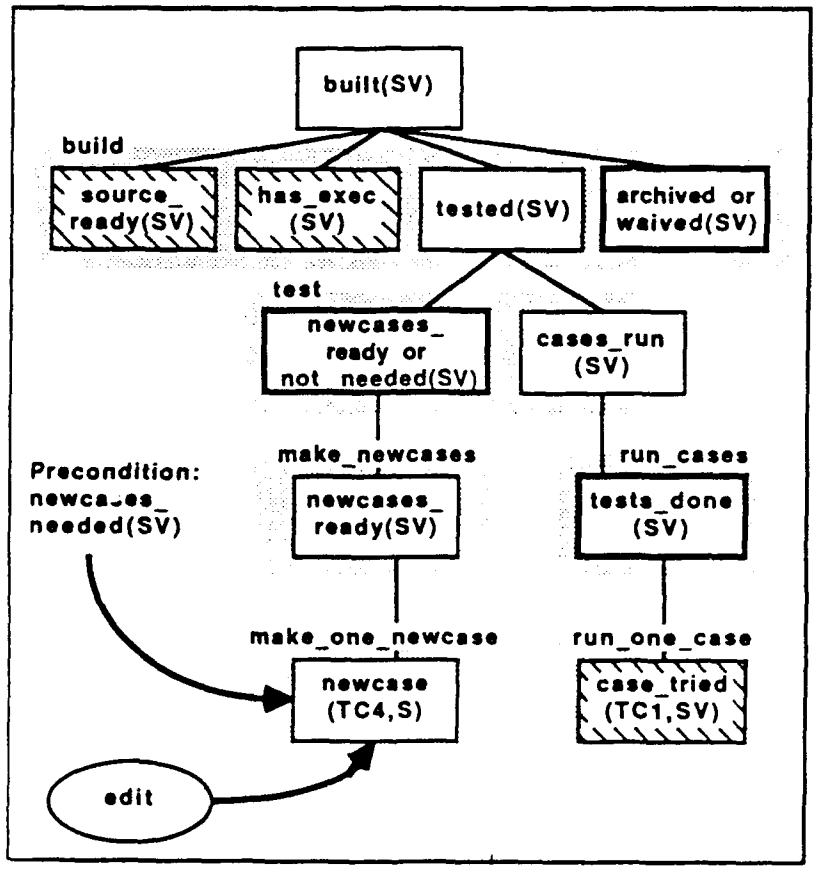

Figure 8: Edit as Making Testcases

proposition that is assumed to be satisfied. Note that if *newcases_needed* had been true with-certainty, then the interpretation would be valid absolutely.

Now consider the interpretation of *edit* as part of getting source code ready; in this interpretation, *edit* starts a new top level plan (shown to the right in Figure 9). Again, *edit* has no preconditions, and *make_newmodule* has no preconditions, but *make_source* has a precondition that the baseline system (on which this new system version is to be based) is built. *Built(SV)* is not true, but there is a plan to achieve it that is in progress. Three assumptions enter into believing that this plan is not finished. They are that new testcases are needed (affecting the first subgoal of *test*), that testing is not done (affecting the iterated subgoal of *run_cases*), and that archiving is not waived (affecting the last subgoal of *build*). Thus, the interpretation of *edit* as part of starting a new *build* plan has low credibility—it conflicts with three current assumptions.

Credibility is a basis for distinguishing the relative likelihood of these two competing interpretations, establishing a clear preference for *edit* as making testcases. Had it been the case that *newcases_needed* was false by-assumption, then the interpretation involving *edit* as making testcases would conflict with one current assumption. In this case, both interpretations violate current assumptions, so the user could be notified that the *edit* action is possibly in error and given the chance to reconsider performing that action. If the user chooses to perform the *edit*, then the interpretation of *edit* as making testcases is (still) the most credible interpretation.

It so happens that discrimination based on the domain-independent heuristic preferring interpretations with the minimal number of top level plans reinforces the preference for edit as making testcases, although in other cases there will be conflicts between credibility and the minimal-plans criterion. The best discrimination decisions will result from combining the evidence from multiple, independent perspectives. Credibility represents a new perspective, derived from deeper modeling of the context of actions.

6.3.1.1.10 Reconciliation

Reconciliation is the process of revising assumptions to make the world state conform to the requirements of an interpretation. This is only necessary when the "best" alternative (considering all available discriminators) still violates a few assumptions about the state of the world, or when other more attractive alternatives were originally chosen but were subsequently disqualified. The standard approach to reconciliation would be to adopt the necessary assumptions (by giving them new justifications), and propagate the consequences through the TMS. This may lead to contradictions (a node and its negation both IN), which can then be resolved via dependency-directed backtracking [76] as implemented for TMS's [28].

The standard approach to reconciliation misses an important opportunity—the opportunity to explain why the desired assumptions should hold. Each node N to be brought IN during reconciliation is simply provided with a new justification *without re-assessing why N failed to be IN* (i.e., without re-assessing any of the labels on nodes appearing in the currently invalid justifications for N). This means that clues that other assumptions are wrong will be ignored. That is, since no attempt is made to bring N IN by making one of its currently invalid justifications valid, no assumptions in the foundations of N that might have been wrong will be revise .

As an example, consider how to explain that testing is in fact done given the situation diagrammed in Figure 5; this is one part of reconciling the interpretation shown in Figure 9. To bring *tests_done(SV)* IN, we must force *not tests_done(SV)* OUT (see justification J10); that

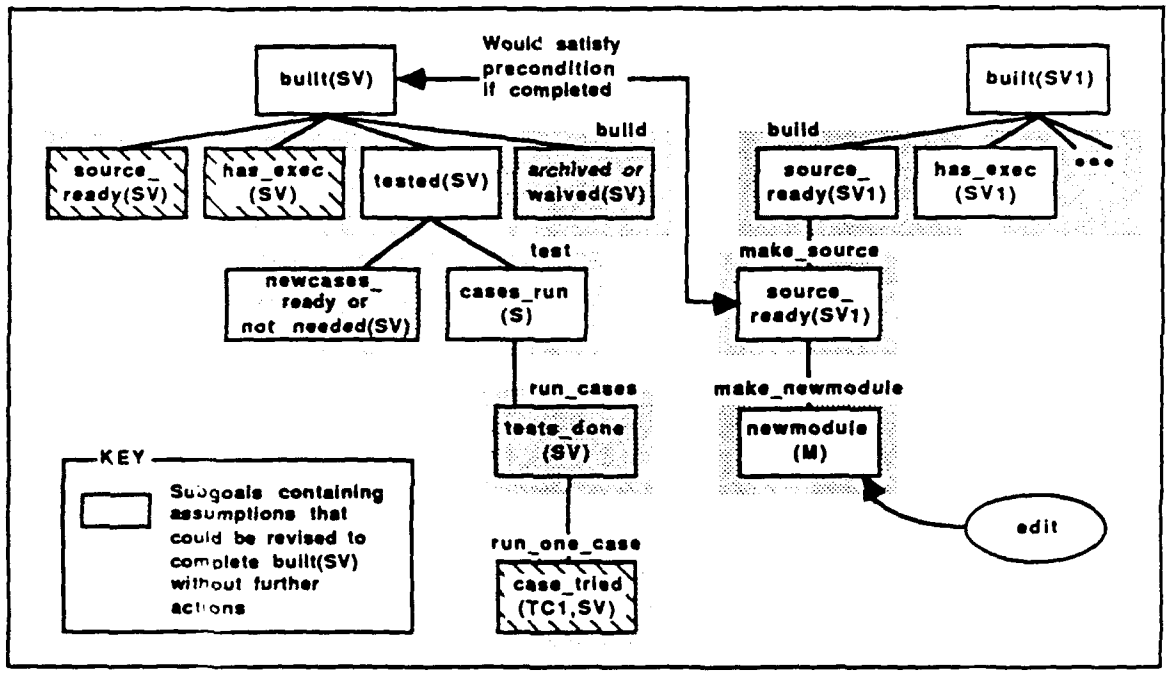

Figure 9: Edit as Preparing Source

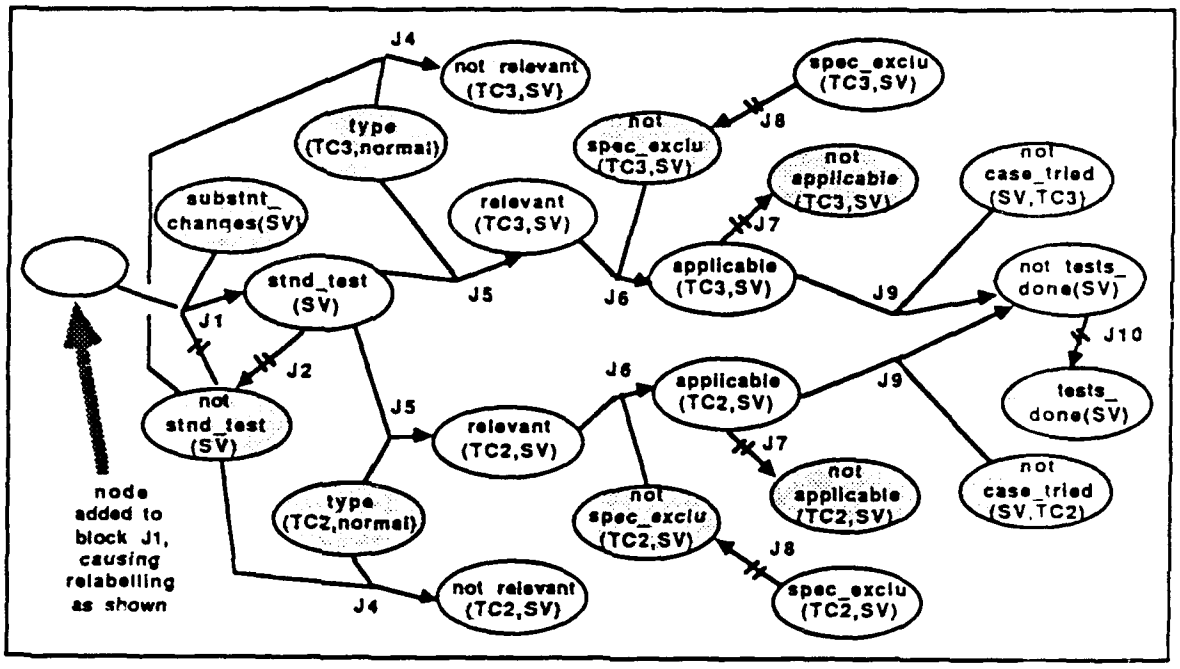

Figure 10: Example of Reconciliation

means forcing OUT both the nodes *applicable(TC2,SV)* and *applicable(TC3,SV)* (covering both instances of justification J9). This can be done in two ways. One way (involving justification J6 and J8) is to bring IN both *spec_excluded(TC2,SV)* and *spec_excluded(TC3,SV)* by adding two "dummy" justifications to support these nodes. The other way (involving justifications J1, J2, J4 and J5) is to block justification J1 that currently supports *standard_test(SV)*; note that this change affects only this instance of rule J1, not any other instances. This latter alternative is heuristically preferred, since it involves one rather than two changes. The result of choosing and installing this alternative is shown in Figure 10.

This approach to reconciliation attempts to achieve a more complete integration of new information into the TMS. The idea of finding explanations for desired beliefs is based on non-monotonic reasoning in the following sense: if a proposition is found to be true, then *typically* one of existing justifications for the proposition should be valid. This type of reasoning is appropriate when all the possible reasons for a proposition being true are represented by justifications in the TMS, which is true for this application, but not for TMS applications in general. Reconciliation can be implemented by extending the algorithm for dependency-directed backtracking to include making invalid justifications valid as well as making valid justifications invalid.

No matter what approach is used for reconciliation, it is possible for reconciliation to fail. This happens when an interpretation depends on assumptions that cannot hold in the current world state. In this event, the interpretation should be rejected.

6.3.1.1.11 Summary

We have shown that missing state information is a barrier to achieving substantive support in an intelligent assistant. In an open world, the only way to exploit additional background knowledge about the context for actions is to introduce uncertainty, in the form of empirical knowledge that makes plausible assumptions about missing data. We have shown how to formalize this additional knowledge (theoretical and empirical) using a TMS to implement non-monotonic reasoning. Then, we have shown how a plan recognizer is affected when the underlying state of the world includes both certainties and assumptions. Assumptions determine the credibility (rather than validity) of competing interpretations, and credibility can be used to make reasoned choices. We have also shown that the plan recognizer will sometimes find its assumptions to be faulty, and we have introduced a new method for revising assumptions in response to this.

6.3.1.2 Implementing an Incremental Hierarchical Plan Recognition System

The task of incremental plan recognition can be computationally expensive, even in a simple domain. Deriving all possible meanings of a particular action, given the limited context of previous actions, can lead to an explosion of competing interpretations, many more than expected at first glance. There are at least two causes. Some interpretations have to be considered viable because unbound parameters are possibly consistent, and multiple top-level goals can be in progress simultaneously. We examine a set of techniques that can be used to contain the proliferation of alternative plan interpretations as soon as possible during the recognition process. These ideas are implemented in a system called GRAPPLE, which is a hierarchical plan recognition system used in an intelligent interface for recognizing user goals from low-level actions.

6.3.1.2.1 Introduction

GRAPPLE recognizes a series of user actions in order of occurrence and incrementally builds reasonable interpretations for the actions. Operators in GRAPPLE are hierarchical and are assumed to be complete. (Research directed at relaxing the assumption that the plan recognizer has complete state information is described in [46] and [44]). GRAPPLE uses the hierarchy to build interpretations through linking of operators (goals of some match subgoals or preconditions of others). By recognizing the possible goals of a user and the alternative ways to achieve them, GRAPPLE identifies the rationale of a proposed action. It then uses its picture of what the user is intending to do in order to to give assistance. Information is provided to the user before a proposed action is actually put into effect. In the case where a proposed action would not be consistent with either the existing state of the world or with previously executed actions, the user is advised of the problem and can specify an alternate action.

The key problem in plan recognition is that the large number of plan derivations implied by even a few actions in a simple world requires careful and quick control of computation. GRAPPLE does aggressive checking to rule out invalid interpretations, it then applies heuristic knowledge to focus on preferred alternatives among those that appear viable. Domain knowledge which is already available is fully exploited. Syntactic and semantic checks on expressions in the operator definitions (preconditions, constraints, and goals) are used to prune out nonviable interpretations. Additional checks, based on the heuristic that the user prefers to act rationally, are used to discard interpretations. These assume that a user will not reachieve a goal already achieved, that a user tends to continue on-going plans, and that a user prefers short plans over long plans. Consequently, only reasonable or plausible interpretations of actions are constructed.

GRAPPLE has been tested in a blocks-world environment and in a simple case of a software development environment [45]. Domain knowledge is contained in the set of operator definitions and associated state schema; the language used for these definitions is based on classical hierarchical planning formalisms [70][85] with some extensions [47]. The state of the world, represented by objects and axioms involving these objects, is implemented using Knowledge Craft.[1] The recognition algorithms themselves are domain-independent. Therefore in order to consider a new or changed domain only the set of operators and the state schema need to be changed.

Plan recognition is performed automatically, without recourse to the user for information beyond the action sequence. At any given time GRAPPLE may be working with incomplete

[1] Knowledge Craft is a trademark of Carnegie Group Incorporated.

information. Variable values for higher level operators are often not yet determined. Because not all the actions have been seen, the system does not know what the future will bring. The only input to the program are the primitive actions and the user determines their order, which may be wrong. In consequence error detection is a critical objective of the system.

A simple example consisting of a sequence of actions in the blocks-world demonstrates the techniques used to control the recognition process. GRAPPLE applies checks as paths are constructed from an action to new top-level goals or pending goals. It continues to check as variables are bound, plans are extended, and action effects are simulated. Some checks are syntactic and semantic: checks for inconsistent bindings, precondition violations, and constraint violations and some checks use the heuristic that interpretations should be plausible: checks for looping plans, redundant plans and on-going plans. All of these checks are applied as soon as possible in order to quickly discard nonviable interpretations and reduce overall computation.

6.3.1.2.2 Operator Definitions

Operators in GRAPPLE are hierarchical. Those used to demonstrate the examples in this section are given in Figure 11. There are three levels of operators. At the lowest level (**stack**, **unstack**) there is no mention of structures, rather just the positions of the blocks. At the middle level (**start-struct**, **extend-struct**, **remove-top-block**, **dismantle-struct**) the concern is with which blocks are in which structures and with a block's role within a structure. At the top level (**make-red-tower**) the concern is with what type of structure exists. GRAPPLE uses the hierarchy to build interpretations.

An operator is composed of a goal, *preconditions*, *subgoals*, *constraints* and *effects*. The linking of operators (goals of some match subgoals or preconditions of others) enables the construction of interpretations. Operators are either primitive or complex. A primitive operator is an explicit user action; it has no subgoals. A complex operator has one or more subgoals and is not an explicit action. In the blocks world **stack** is a primitive operator and **make-red-tower** is complex. It is useful to look at complex operators as referring to higher level concepts and activities. The purpose of complex operators is to decompose more complicated goals into simple ones, allowing a hierarchical view of domain activities.

The preconditions of an operator define the state from which the operator can legally be executed. Consequently all preconditions of an operator must be true simultaneously before any action is taken to satisfy the operator's goal or subgoals. This implies that the preconditions of a complex operator must all be true before any primitive action in the expansion of one of its subgoals begins. There are two kinds of preconditions, normal and static. A normal precondition can intentionally be satisfied by taking actions while a static precondition can not. A precondition which is not explicitly defined as static in the operator definition is understood to be normal.

By using subgoals a complex operator is decomposed into subproblems, each of which must be satisfied before the effects of the operator are achieved. For example, **make-red-tower** is a complex operator having two subgoals. It is divided into the subproblems of starting the tower and finishing it. In the case when an operator has more than one subgoal, the order in which subgoals should be satisfied is determined solely by the state of the world and the preconditions of the operators being used to satisfy the subgoals. The only restriction is that all subgoals which are labeled "final" must be true simultaneously to enable installing of the effects clause.

Constraints restrict the bindings in the operator. They must not be violated from the time the

operator's preconditions are true until the time its effects are posted. Effects are the changes to the data base (the "world") which result from executing an action or from completing a complex operator. New objects can be created, attribute values can be set, new predicates can be added, and old ones deleted. Further, an effect can be made conditional on the state of the world. Not only do the effects of an operator cause its goal to be true, but often additional changes are made in the state of the world. These are side effects.

The recognition algorithms regard the goal of an operator as the main purpose for an operator's execution. The plan network which is built by matching operator goals to subgoals and normal preconditions of other operators is central to the recognition process. In particular, the purpose of an operator might be seen as the achievement of a subgoal of a second operator which might in turn be satisfying the precondition of another operator. The reason, then, of carrying out a series of actions can be to satisfy the goal of some top-level operator; a top-level operator is one whose goal does not satisfy a precondition or subgoal of any other operator.

6.3.1.2.3 The Recognition Cycle

After GRAPPLE is first initialized, an initial world state is input to the program. The actions are then processed one at a time. Before an action is actually completed and in consequence the state of the world is changed, GRAPPLE builds interpretations of the proposed action. These are used by the program to decide whether or not an action should be taken and what, if it should, its purpose would be. A context is a world view. Different contexts provide alternative world views; each is a separate data base state. An interpretation for an action or sequence of actions is a tree of operators where top-level operator goals are linked to primitive actions. When all the variables of the linked operators in an interpretation are bound to values, the interpretation is complete. At any given time during the recognition process there might be a number of alternative interpretations for the sequence of actions, all with operator variables partially bound. When no interpretations for an action exist, the user is advised to take another action. When many exist, focusing decisions are made to pick preferable interpretations.

GRAPPLE can consider actions leading to more than one top-level goal simultaneously. However in the case where a particular action leads to two or more top-level goals at the same time only one is the "purpose" of the action in a particular interpretation and the others are side effects.

A broad outline of the plan recognition cycle is given below.

```
Initialise GRAPPLE: establish links between operators.
While: there is a proposed new action.
    For each active context:
        Find all valid possible interpretation paths from action to top-level goal.
        If: No interpretations in context, go to next active context.
        Else:
            For each interpretation in context:
                    Make new child context and instantiate operators.
                    Test extended interpretations for inconsistent bindings.
                    Test for possible constraint or precondition violation.
                    If interpretation invalid:
                        refute context.
```

go to next interpretation.

Else:

Assert effects of the action in the interpretation.

Evaluate all task preconditions, constraints, and subgoals.

Assert effects of enabled complex operators.

Monitor constraints, looping and redundancy.

If interpretation invalid: refute context.

Go to next interpretation.

When no more interpretations in context:

Focus: prefer child contexts with extended interpretations.

Parent context is superseded.

Go to next active context to interpret action.

When no more active contexts:

Further focus: prefer child contexts where all actions are interpreted as extensions.

If action can be interpreted:

Establish new group of active contexts.

If action can not be interpreted in any context.

Inform user.

Restore previous group of active contexts.

Get new action.

6.3.1.2.4 Initializing GRAPPLE

GRAPPLE is initialized by establishing the links between the operators in the library. A graph showing the links between operators is shown in Figure 1. For simplification the graph does not show all links. For example, **remove-top-block** also links to **unstack** through its normal precondition, (clear ?x). For each logical expression that is a subgoal or normal precondition of an operator, alternate achievers are computed. An alternate achiever is an operator whose goal, when true, achieves the expression; a set of variable mappings is determined with each alternate achiever. The operator **stack** is an alternate achiever of the subgoal (on ?top-block ?base-block) of **start-struct** and the mapping is ?x → ?top-block, ?y → ?base-block. Two things can be observed: only part of an operator's goal need match the condition for it to satisfy the expression and an operator's goal can achieve a condition in more than one way. For example, **dismantle-struct** achieves the precondition (clear ?y) of **stack** with a mapping ?base-block → ?y and a mapping ?top-block → ?y. In this case there are two alternate achievers involving **dismantle-struct**. In the current implementation of GRAPPLE, when an operator's goal only partially satisfies a subgoal or precondition of a second operator, a link is not formed.

Once links are established they are used to generate paths from primitive actions to top-level goals. When a user action is proposed and its preconditions and constraints are not violated, possible interpretations are generated. A possible interpretation is a path, linked through goals and logical expressions from the action to a top-level goal.

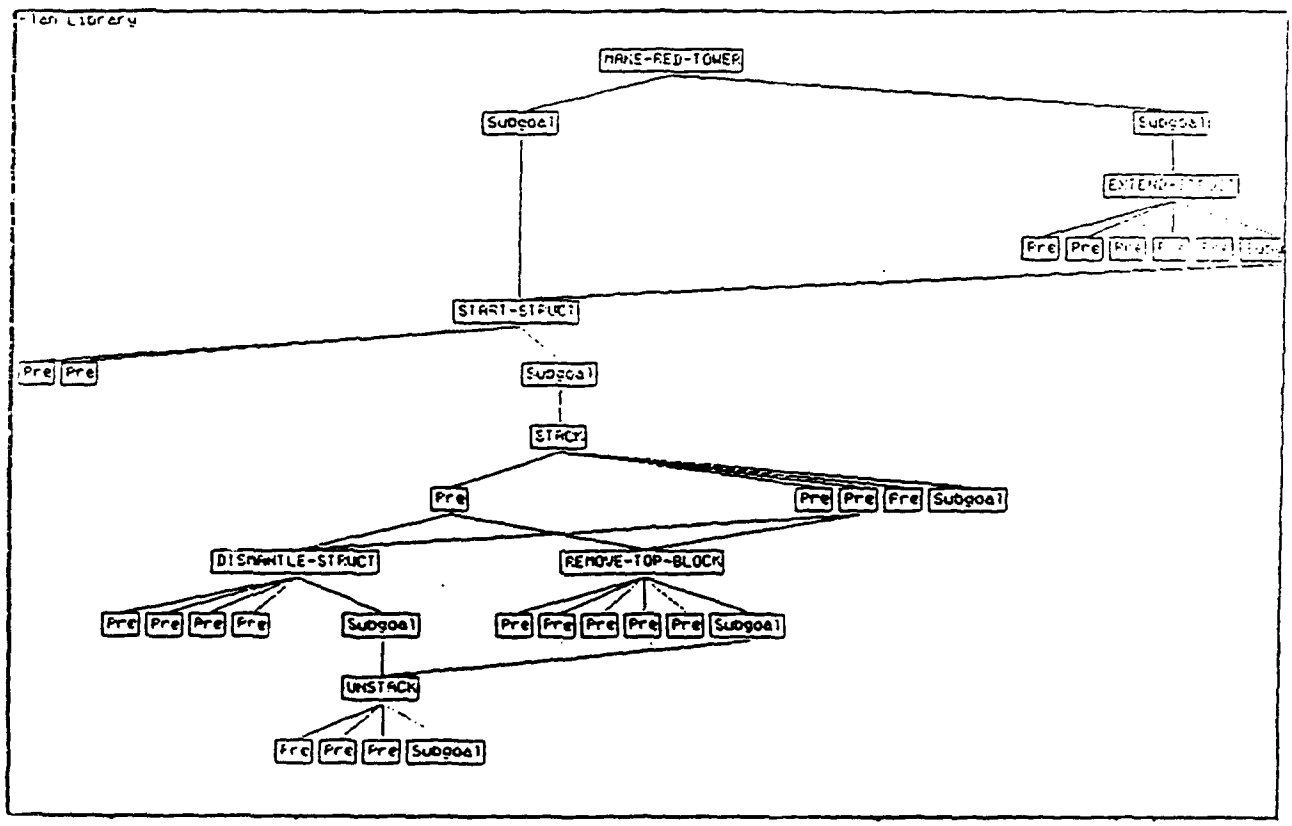

Figure 1: The Plan Library

6.3.1.2.5 Limiting Possible Interpretations When an Action is Proposed

The strategy used in GRAPPLE is to limit the number of possible interpretations by aggressive checking to throw away those that can not be valid. GRAPPLE does not consider possible interpretations when it can predetermine that a higher level operator's preconditions or constraints are violated, that an already satisfied goal will be resatisfied or that endless cycles exist. Checking starts as the interpretations are generated; the viable interpretations are then instantiated and checked further as described in section 6. In the identification of all possible paths from the action to the top-level goal an interpretation is discarded if any of the conditions listed below are violated. These conditions are tested as each operator is linked upward. The values of the variables used in the tests are those given as parameters to the action; these values are propagated up through the links. For example, the action (unstack cube3 cube1) leads to **dismantle-struct** with ?top-block=cube3 and ?base-block=cube1. The construction of a particular path upward is discontinued as soon as any operator fails the tested conditions. The result is that the computation involved in generating possible interpretations, and the number of these interpretations, is greatly reduced.

Tested syntactical and semantic conditions in the bottom-up generation of possible paths:

- If an operator is linked, either directly or indirectly, to the action through one of its subgoals then none of its preconditions can be false. The truth of a precondition is determined by querying the data base. If all the variables in the precondition are bound to values, the precondition must be true. For example, if in the initial state cube3 is on cube1, then the goal of the action, (unstack cube3 cube1) satisfies the subgoal of **dismantle-struct**, the variable mapping is cube3 → ?top-block and cube1 → ?base-block. If any of the preconditions of **dismantle-struct** are false with this variable mapping, such as cube3 not being in a structure, then no possible path can be derived which starts by **unstack** linking to **dismantle-struct** in this way.

- No constraints of an operator are violated. Again the bindings are those taken from the action (as propagated up through the path link via the mappings). Since there are no constraints on **dismantle-struct**, the test will not apply. If cube1 eventually maps into ?first-cube of **make-red-tower**, then it must be red to satisfy the top-level constraint, (color ?first-cube red).

Tested conditions, based on the heuristic that the user prefers to be rational, in the bottom-up generation of possible paths.

- No goal of an operator having all its variable values bound from the parameters of the action when propagated upward is already true in the current state of the world. The user's action in such a case would serve no purpose since the intent of the action is seen as satisfying an already satisfied goal. If a goal is already true, any interpretation which reachieves it should be discarded.

- No linked precondition or subgoal of an operator is already true in the current data base state. For example the path starting as follows (unstack cube3 cube1) leading to the subgoal (not (on ?top-block ?base-block)) of **dismantle-struct** leading to the precondition (clear ?y) of **stack** with cube3 = ?top-block = ?y will be discarded because cube3 is already clear in the current state.

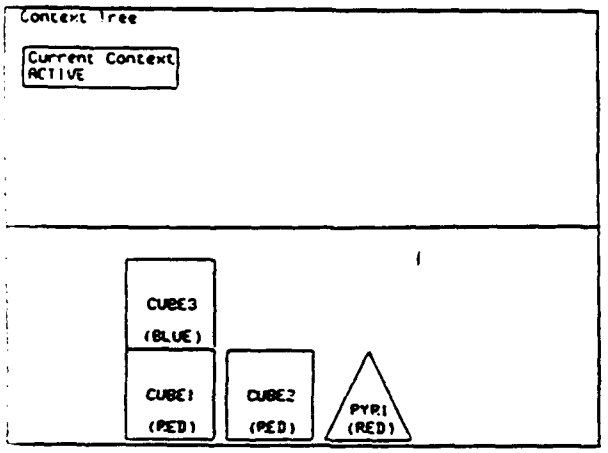

Figure 2: The Initial State and Context

- No loops exist in the interpretation. No operator is revisited on a path with identical variables bound to the same values or with identical variables unbound. This insures that possible interpretations in which cycles occur are not considered. For example, the operator (stack cube2 ?y) can not occur more than once in a path, although both (stack ?x ?y) and (stack cube2 ?y) could appear.

In order to inhibit endless searching a given cutoff level or maximum number of operator links allowed is input to GRAPPLE. This is based on another "rationality" heuristic: that the user will not prefer long paths to top-level goals when shorter ones exist. There are some instances when an action taken in a given data base state can not lead to any top-level goal through a reasonable number of links. When this occurs or when there are no valid possible interpretations, the action can not be recognized and the user is so advised.

6.3.1.2.6 Exploiting Available Constraints

The process of limiting the derivation of interpretations and checking those that remain is best understood by looking at a particular example. The illustrations in the figures are actual output from the program. Figure 2 shows the initial state of the world for the example. There is only one context, the initial world state, in which to interpret the first action. The first action is (unstack cube3 cube1). Four possible interpretations leading to the goal of completing a red tower are derived. Nineteen possible interpretations are discarded while testing the conditions (described in section 5) in the process of finding these potentially acceptable paths.

The four interpretations are instantiated before further analysis is done to see if there are additional reasons to discard them. Each interpretation is tested in turn. For each, the original parent context is copied to a new context. Every operator of the interpretation within the new context is instantiated as a separate task (instantiated operator) and its preconditions and subgoals are expanded. The original parent context, replaced by new ones, is *superseded.* When an interpretation within a context is found to be invalid, the context is *refuted.* In the example, there are four ways to interpret the first action in the parent context ; four new contexts are created. Figure 3 shows the context tree after the instantiation of the four interpretations of the first action. Figure 4 shows the instantiation of an interpretation in a particular context.

As stated previously, preconditions of operators linked to the action through a subgoal must

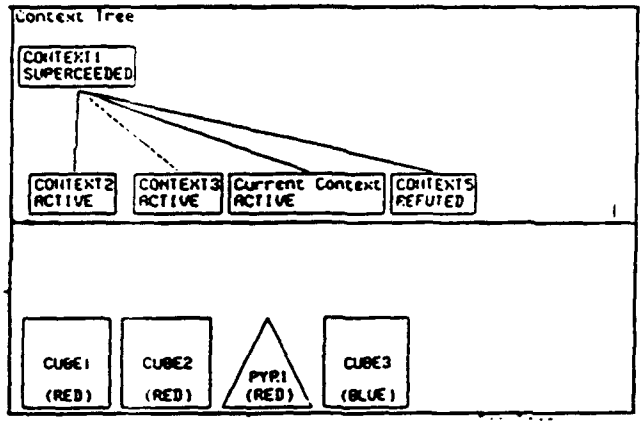

Figure 3: State and Contexts after the action (unstack cube3 cube1)

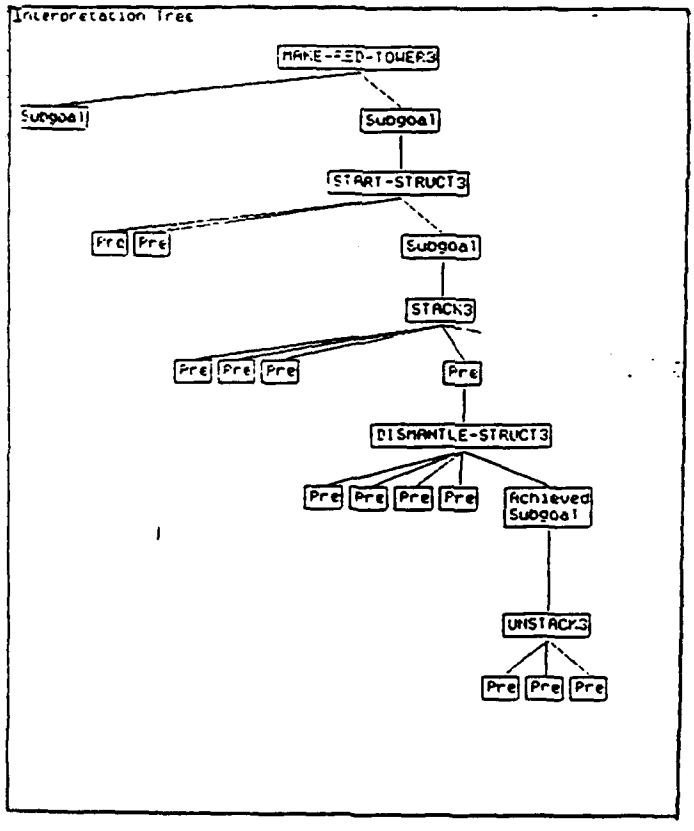

Figure 4: Interpretation in Context2 after the action (unstack cube3 cube1)

be true for the interpretation to be valid. No interpretations with operators having false preconditions and bound variables survived the derivation, but there may exist operator preconditions with unbound variables which are false because values do not exist for those variables in the existing data base state which will make the expressions true. Because it is necessary that all preconditions of a complex operator be true before any action is taken which leads to one of its subgoals, the program allows the binding of variables through preconditions from the existing state of the data base when such an action is proposed. Any variable values which are found in this way must not violate the constraints of the operator. If a variable is propagated up through links of the interpretation and is included in other operator constraints or preconditions which must also be true (when the operator is linked through a subgoal), then these expressions further the restrictions on the bindings. The result is a query to the data base with a set of logical expressions which must all be true simultaneously. This query constrains the search for plans in three ways. It throws out invalid interpretations, it binds variables when the bindings are unique, and it provides constraints on variables when the bindings are not unique.

In the first interpretation of the example (unstack cube3 cube1) satisfies the subgoal of **dismantle-struct** (all its preconditions are true with bound variables from the action); **dismantle-struct** satisfies the normal precondition, (on-table cube1) (clear cube1), of **stack**; **stack** leads to the subgoal (on cube1 ?base-block) of **start-struct**; and **start-struct** leads to a subgoal of **make-red-tower**. In the solution all the preconditions of **start-struct** must be true for the interpretation to be valid. The data base is queried with the two precondition clauses of **start-struct** and the relevant constraint clauses of **make-red-tower**. As a result the variable ?base-block of **start-struct** is bound to cube2 from the data-base; this is the only value in the data-base which is on the table, not equal to cube1 and satisfies the constraints (?first-cube being a cube and red) of the operator **make-red-tower**.

When there are no values for the variables in the data base which satisfy the query, the precondition is false. This is what happens in the fourth interpretation which is instantiated in context5 (see Figure 3). The path from the action, (unstack cube3 cube1), is **unstack** → **dismantle-struct** → **stack** → **extend-struct** → **make-red-tower**. **Extend-struct** leads to the subgoal (add-pyramid) of **make-red-tower**. The variables ?top-block and ?base-block of **extend-struct** are both unbound. The data base is queried using the preconditions of **extend-struct** and relevant constraints of **make-red-tower**. The query fails because the expressions can not be solved simultaneously. This is because ?lower-block and ?base-block both are bound to cube1 which violates the precondition that they can not be equal. The interpretation is discarded and the context refuted. Notice context5 in Figure 3 is refuted. When more than one value can be bound to a variable, the alternative possible values of the variable are added as a constraint on the task. The effects of any task which affect or use such a variable are not activated until one of the multiple values is actually forced on the interpretation by a subsequent action. The additional constraint on the variable may restrict the possible extensions of the interpretation.

By exploiting the constraints in the example, GRAPPLE determined that there were only three possible interpretations for the proposed first action.

Finally it should be emphasized that GRAPPLE binds variables in only three ways: from an overt action and the linking of operators, from asserting effects which create new variable values, and from a query to the data base with expressions which must be true.

6.3.1.2.7 Monitoring Constraints, Looping, and Redundancy with Data Base Changes

Some interpretations may be discovered to be invalid only after the effects of the action are simulated. For instance, a side effect of an operator could cause the violation of a precondition or constraint of another operator (upward on the path).

The state of the world actually changes in each *active* (neither *superseded* or *refuted*) context when the effects of an action or operator are asserted. (In GRAPPLE all the contexts which are active are carried forward). Afterwards every task (instantiated operator) in the context is then processed to see if previously unsatisfied preconditions or subgoals are now satisfied due to the data base changes. Tasks exist in varying stages of completion— :trying-to-achieve-preconditions, :trying-to-achieve-subgoals, or :accomplished. When all of the tasks in a context are accomplished, the top-level goal is achieved. If the state of a task is :trying-to-achieve-preconditions, the preconditions having bound variables are tested. When all of a task's preconditions are true simultaneously, the state of the task is changed to :trying-to-achieve-subgoals. If the state of a task is :trying-to-achieve-subgoals, the subgoals having bound variables are tested. Once the subgoals are satisfied and the truth of all the task constraints on the interpretation path is established, the effects of the task are asserted. The effects of complex operators are automatically asserted when its subgoals become true and that primitive operators are overt actions which must be taken by the user.

Redundant plans or actions and looping may be discovered as variables are bound. One of the ways a context can be refuted is by the discovery that there exists two different tasks in the context which are instantiations of the same operator and which either have matching bindings or have the set of bindings for one task subsumed by the bindings of the other. This means that these two tasks are combined in some other context to form one interpretation or that looping occurs. Notice that it is possible that a precondition or subgoal can become true as a result of asserting the effects of a task even though the goal of the task in the interpretation is not to satisfy that particular condition. No variables are bound during the process of checking the truth of preconditions or subgoals of a task; an expression is true only if its variables are bound.

There is an additional reason to discard a context when the data base is altered. Changes in the world state may violate operator constraints. Therefore, when any constraint is found to be false while processing the tasks, the context is refuted.

Consider the example. There are three active contexts for the first action after the effects are posted in each (see Figure 3). The interpretation in context2 is illustrated in Figure 4. After the action (unstack cube3 cube1) the tasks unstack3 and dismantle-struct3 are completed; and the preconditions for stack3 are true. The task, start-struct3, is in the state :trying-to-achieve-subgoals as is the task make-red-tower3. Three subgoals need to be satisfied in order to achieve the top-level goal. If all the child contexts (context2, context3, context4) are refuted the primitive action can not be recognized and the user is is so advised. This does not happen in the example. In the example none of the simulated effects of the first action caused GRAPPLE to discard an interpretation. The action is taken in each active context.

6.3.1.2.8 Narrowing Interpretations with Additional Actions

Additional actions reveal more of the plan and provide information which is used to discard existing interpretations.

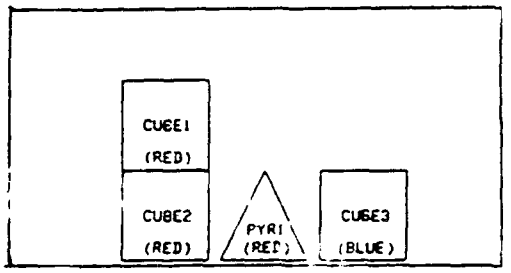

Figure 5: State after the action (stack cube1 cube2)

After the first action is processed in all active contexts GRAPPLE asks the user for the next action. This action is considered in each active context in turn. In the example (stack cube1 cube2) is the second action; it is interpreted in context2, context3, and context4. Possible interpretation paths are derived and discarded as before. There is a difference though and that is: a path can lead to an operator corresponding to an instantiated task which is pending (not yet complete) in the context as well as to a top-level operator. For example: since stack3 is a pending task in context2, a possible interpretation leads to **stack**. In each context of the example four possible interpretations out of eight survived the derivation.

Each surviving candidate interpretation can either be an extension of an existing interpretation or, if it leads to a top-level goal, the start of a new interpretation in the given context. An interpretation which does not lead to a top-level operator must necessarily be an extension. An interpretation which does lead to a top-level operator might be an extension if it leads to an unsatisfied precondition or subgoal of a partially satisfied top-level task. Alternatively it might be *the start of a new interpretation* to a top-level goal.

There may be reasons to reject an extended interpretation before tasks are instantiated. When the bindings of the new action are inconsistent with the existing bindings of the top-level task, when the new bindings will cause a constraint violation in the existing interpretation or when a precondition of a task linked through its subgoal is violated with the new bindings added, the interpretation is not considered as an extension. For example, in both context4 and context3 two of the possible interpretations, one leading to **stack** and another leading to ***start-struct*** are thrown out due to inconsistent bindings.

For each remaining candidate interpretation, whether it is an extension or a new interpretation, the active context is copied to a new context along with the pending tasks. The parent context is superseded. The analysis done in each surviving context is the same as is described for the first action. Contexts are refuted if necessary preconditions can't be true. Pending tasks are processed, effects are asserted and further tests are done to refute contexts.

In building new interpretations, there is a chance that redundant plans evolve. In cases where an interpretation can be viewed as both an extension and the start of a new interpretation, a test is done to see if the two interpretations are in fact the same. If the top-level goals are identical and the variable bindings of the new interpretation are subsumed by those of the extension, the context corresponding to the start of a new interpretation is refuted. This is what happens when the final action of the example is interpreted as the start of a new plan in context9 (described in the next section).

Looking at the example, the action (stack cube1 cube2) can be interpreted as an extension of the existing interpretation in context2. The bindings are consistent and no constraints or necessary preconditions are violated. Those interpretations where bindings are inconsisten , con-

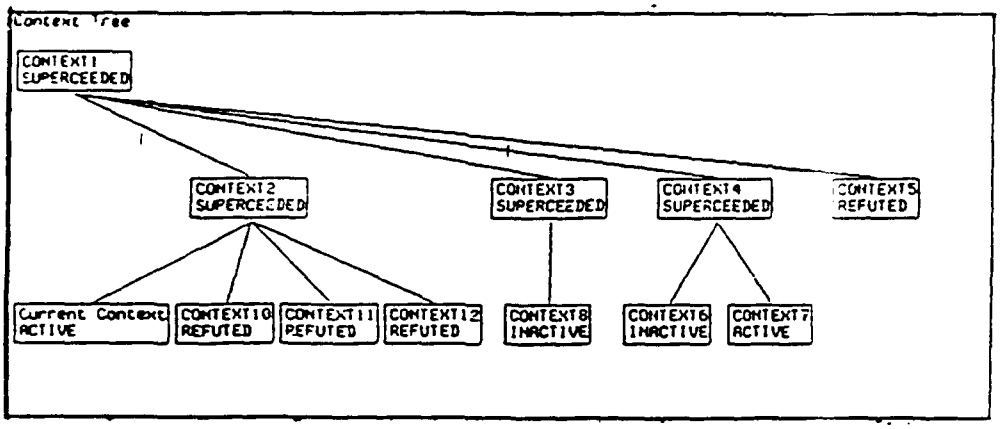

Figure 6: Contexts after the action (stack cube1 cube2)

straints are violated or preconditions violated are not pursued. The effects of **stack** are asserted, accomplishing stack11 and start-struct10 and achieving a subgoal of make-red-tower10. Figure 7 shows the task tree for the interpretation. Of the possible interpretations of the action in context2 three child contexts, context10, context11, and context12 are refuted because two instantiations of the operator **stack** exist with identical bindings in the context. Figure 6 represents the context tree after the second action. In context3 the action (stack cube1 cube2) can not be interpreted as an extension because the bindings are inconsistent. The only interpretation allowed in this context is one leading to a new top-level goal of making a red tower.

As it stands the program does not resolve the problem which arises when achieved preconditions or subgoals of a task are violated prematurely by asserting the effects of later actions. In this case a decision should be made whether to go ahead with the action or retract it. If an action is retracted, it must be retracted in all contexts. On the other hand when the action is taken, previously satisfied conditions which are violated must be resatisfied by future actions. The existence of conditions in a context that are violated in this way could also be used to focus the recognizer. The focusing heuristic is: the user prefers plans in which conditions do not have to be resatisfied.

6.3.1.2.9 Focusing on Extended Interpretations

Another way of controlling the plan recognition process is to focus on selected interpretations while saving others for future development. When a new user action is proposed, GRAPPLE first focuses on interpretations where the action extends an existing plan. The program prefers to assume that the user has a plan or set of plans in mind [17]. Irrefuted contexts in which an action can be interpreted as extending an existing plan are made active; other irrefuted contexts are made inactive. An *inactive* context, neither refuted or superseded, is saved for future processing. In the example, context8 is made inactive because (stack cube1 cube2) can not be interpreted as an extension, but it can be interpreted as the start of a new plan. When there are no contexts in which the action can extend a plan, all contexts where the action starts a new plan are made active.

When there are no contexts in which an action can be interpreted, the action can not be recognized. Consider the situation shown in Figure 5. The second action of the example has

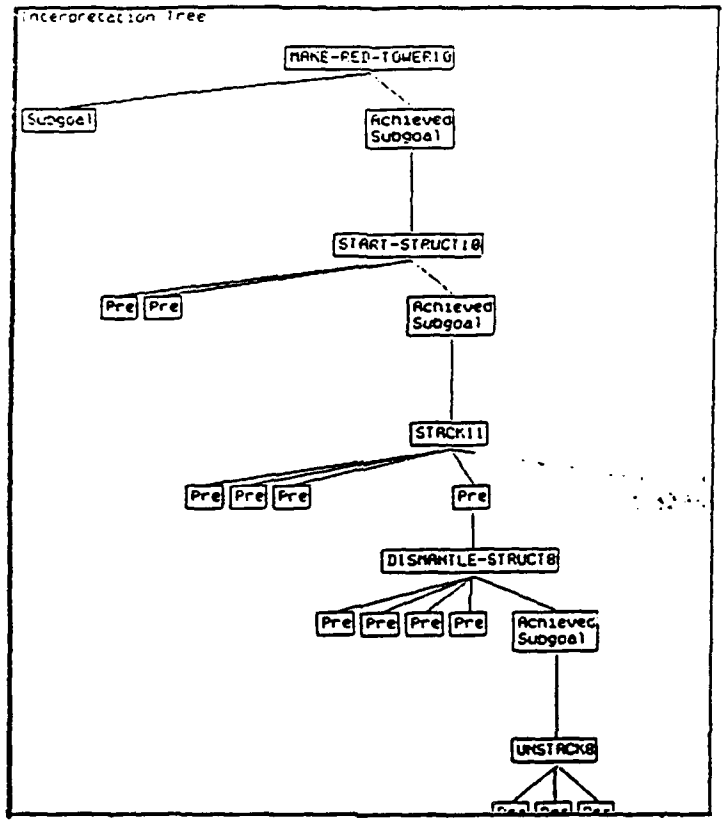

Figure 7: Interpretation in Context9 after the action (stack cube1 cube2)

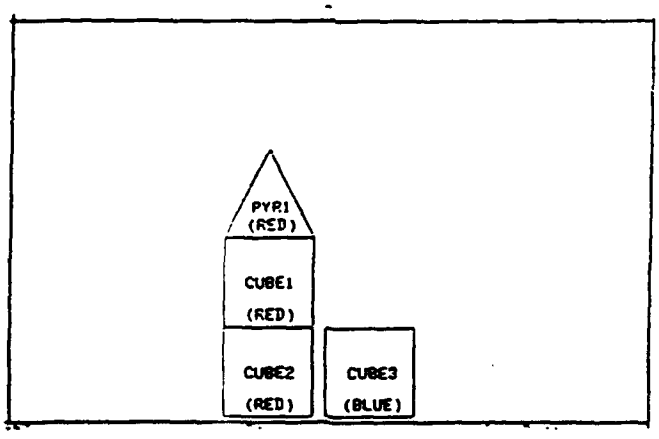

Figure 8: State after the action (stack pyr1 cube1)

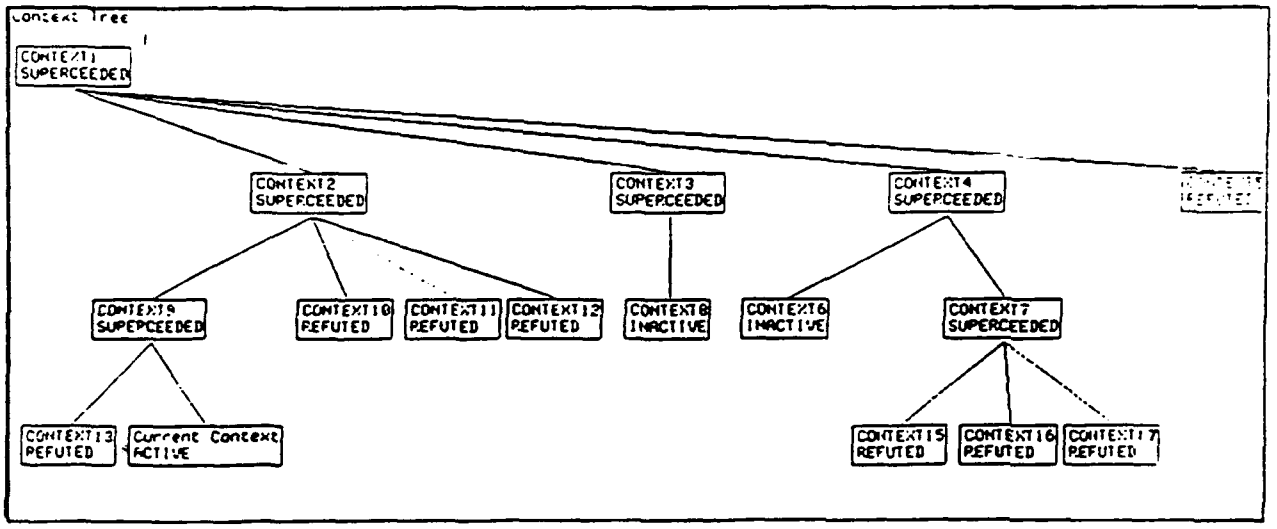

Figure 9: Contexts after the action (stack pyr1 cube1)

been processed. If the user then proposes an action such as (stack pyr1 cube3), GRAPPLE can not find a way to interpret the action in any active context. It then makes all inactive contexts active, processes the unprocessed actions in the newly activated contexts and tries again to interpret the given action. If the action still can not be recognized, the user is informed. The interpretations and contexts are then changed back to the state that existed before the original action was proposed. The above scenario is not illustrated in the example.

The third action of the example is (stack pyr1 cube1), as shown in Figure 8; it is processed in the active contexts, context9 and context7 and is added to the list of unprocessed actions in the inactive contexts, context8 and context6. All the candidate interpretations in context7 are found to be invalid. The action, interpreted as an extension, in context9 achieves the subgoal of extend-struct3 which achieves the second subgoal of make-red-tower17. The action, interpreted as the start of a new plan, in context9 gives a redundant interpretation. The top-level goal is accomplished in context14. The final context tree is shown in Figure 9 and the final interpretation in Figure 10. The plan is recognized.

6.3.1.2.10 Multiple Plans

The ability to limit possibilities is even more valuable when there are multiple plans being carried out in parallel. GRAPPLE can recognize more than one plan simultaneously. When a top-level operator, **make-blue-tower**, is added to the set of operators, actions for building a red tower and a blue tower can be interleaved and the program recognizes the purpose of the actions. For instance, the first action of unstacking a blue cube from a red cube will have an additional interpretation of dismantling the original structure in order to build a blue tower.

6.3.1.2.11 Conclusion

We have found that incremental hierarchical plan recognition is more computationally expensive than expected. Even in a simple domain the task can quickly get out of control.

In a simple example nineteen possible interpretations for the first action are reduced to three

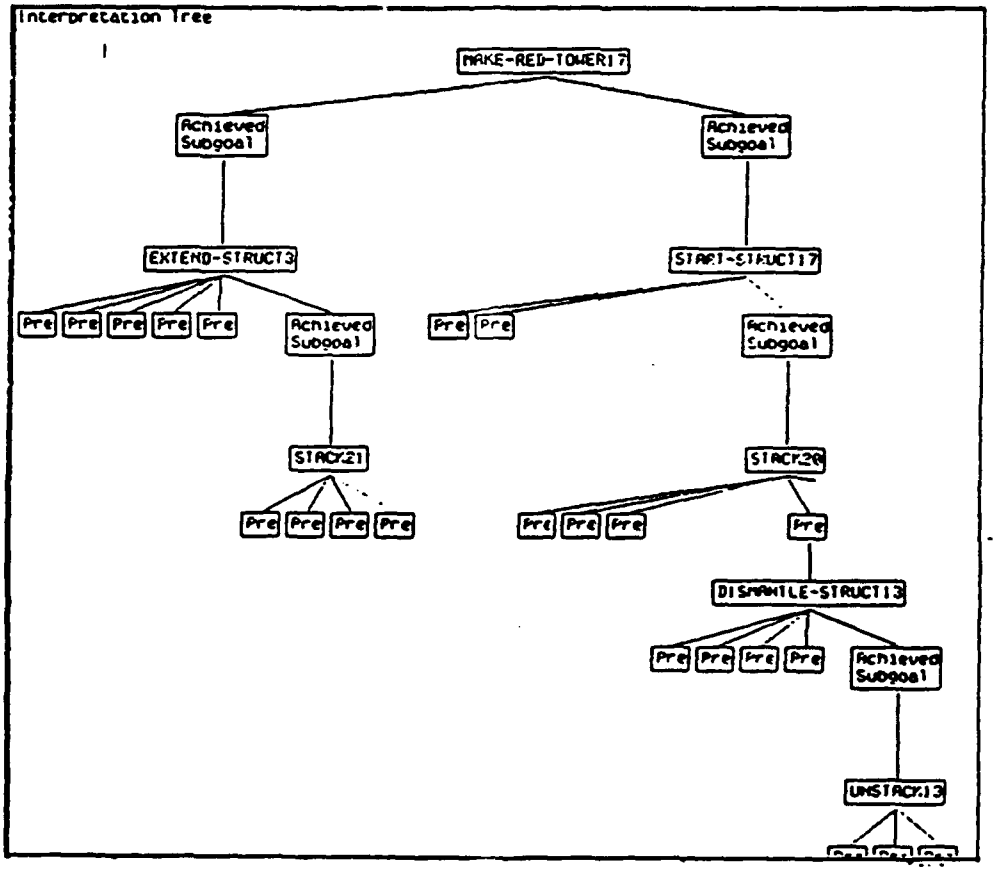

Figure 10: Interpretation in Context14 after the action (stack pyr1 cube1)

(of these twelve are discarded based on the heuristic that the user will behave rationally and interpretations are plausible). Checking and focusing reduces the number of interpretations for the first followed by the second action from twenty-four to two (of these eleven are discarded based on the "rationality" heuristic). Finally, GRAPPLE finds only one way to interpret the sequence of three actions out of ten possible interpretations (of these two are discarded based on the "rationality" heuristic).

Non-trivial mechanisms are required to control the combinatorial explosion of possible interpretations. There is a need to exploit constraints quickly and to focus on reasonable or plausible interpretations. With the techniques described here incremental plan recognition becomes tractable.

Figure 11: Example Blocks World Operators

unstack

```
(defplan unstack (?x ?y)
(goal (not (on ?x ?y)))
(precondition on-x-y (on ?x ?y) :static)
(precondition clear-x (clear ?x))
(precondition not-eq (not (= ?x ?y)) :static)
(subgoal (primitive))
(effects (add (clear ?y))
                    (add (on-table ?x))
                    (delete (on ?x ?y))))
```

remove-top-block

```
(defplan remove-top-block (?top-block ?block2 ?structure)
(goal (and (clear ?top-block) (on-table ?top-block)
                    (clear ?block2)))
(precondition not-on-table (not (on-table ?block2)) :static)
(precondition in-struct1 (in ?block2 ?structure) :static)
(precondition in-struct2 (and (in ?top-block ?structure)
                    (top ?structure ?top-block)) :static)
(precondition on-blk (on ?top-block ?block2) :static)
(precondition not-eq (not (= ?top-block ?block2)) :static)
(subgoal unstack-blks (not (on ?top-block ?block2)) :final)
(effects (delete (top ?structure ?top-block))
                    (delete (in ?top-block ?structure))
                    (add (if (old (not (base-block ?structure ?block2)))
                                        THEN (top ?structure ?block2)))
                    (delete (if (old (base-block ?structure ?block2))
                                        THEN (in ?block2 ?structure)))
                    (delete (if (old (base-block ?structure ?block2))
                                        THEN (base-block ?structure ?block2)))
                    (set (type-struct ?structure unknown))))
```

dismantle-struct

```
(defplan dismantle-struct (?top-block ?base-block ?structure)
(goal (and (clear ?base-block) (on-table ?base-block)
                    (clear ?top-block) (on-table ?top-block)))
(precondition in-struct1 (and (in ?base-block ?structure)
                    (base-block ?structure ?base-block)) :static)
(precondition in-struct2 (and (in ?top-block ?structure)
                    (top ?structure ?top-block)) :static)
(precondition on-blk (on ?top-block ?base-block) :static)
(precondition not-eq (not (= ?top-block ?base-block)) :static)
(subgoal unstack-blks (not (on ?top-block ?base-block)) :final)
(effects (delete (top ?structure ?top-block))
                    (delete (in ?top-block ?structure))
                    (delete (in ?base-block ?structure))
                    (delete (base-block ?structure ?base-block))
                    (set (type-struct ?structure unknown))))
```

stack

```
(defplan stack (?x ?y)
(goal (on ?x ?y))
(precondition not-com3 (and (on-table ?x) (clear ?x)))
(precondition not-pyr-y (not (pyramid ?y)) :static)
(precondition clear-y (clear ?y))
(precondition not-eq (not (= ?x ?y)) :static)
(subgoal (primitive))
(effects (add (on ?x ?y))
                    (delete (clear ?y))
                    (delete (on-table ?x))))
```

start-struct

```
(defplan start-struct (?base-block ?top-block ?structure)
(goal (and (on ?top-block ?base-block)
                    (top ?structure ?top-block)
                    (base-block ?structure ?base-block)
                    (in ?top-block ?structure)))
(precondition on-table (on-table ?base-block) :static)
(precondition not-eq (not (= ?base-block ?top-block)) :static)
(subgoal stack-blks (on ?top-block ?base-block) :final)
(effects (new (?structure structure))
                    (add (top ?structure ?top-block))
                    (add (in ?top-block ?structure))
                    (add (in ?base-block ?structure))
                    (add (base-block ?structure ?base-block))
                    (set (type-struct ?structure unknown))))
```

extend-struct

```
(defplan extend-struct (?lower-block ?top-block ?base-block ?structure)
(goal (and (on ?top-block ?lower-block)
                    (top ?structure ?top-block)
                    (in ?top-block ?structure)
                    (base-block ?structure ?base-block)))
(precondition in-struct (in ?lower-block ?structure) :static)
(precondition in-struct1 (base-block ?structure ?base-block) :static)
(precondition not-eq1 (not (= ?base-block ?top-block)) :static)
(precondition not-eq2 (not (= ?lower-block ?base-block)) :static)
(precondition not-eq (not (= ?lower-block ?top-block)) :static)
(subgoal stack-blks (on ?top-block ?lower-block) :final)
(effects (add (top ?structure ?top-block))
                    (add (in ?top-block ?structure))
                    (delete (top ?structure ?lower-block))
                    (set (type-struct ?structure unknown))))
```

make-red-tower

```
(defplan make-red-tower (?structure ?first-cube ?second-cube ?pyramid)
(goal (redtower ?structure))
(subgoal build-foundation (and (on ?second-cube ?first-cube)
                    (in ?second-cube ?structure)
                    (base-block ?structure ?first-cube)) :final t)
(subgoal add-pyramid (and (in ?pyramid ?structure)
                    (base-block ?structure ?first-cube)
```

```
                (on ?pyramid ?second-cube)) :final t)
(constraints (pyramid ?pyramid) (cube ?first-cube) (cube ?second-cube)
                (color ?pyramid red) (color ?first-cube red)
                (color ?second-cube red))
(effects
                (set (type-struct ?structure tower))
                (set (color ?structure red))))
```

6.3.1.3 Planning for the Control of an Interpretation System

Interpretation is a complex and uncertain process which requires sophisticated evidential reasoning and control schemes. We have developed a framework which models interpretation as a process of gathering evidence to manage uncertainty. The key components of the approach are a specialized evidential representation system and a control planner with heuristic focusing. The evidential representation scheme includes explicit, symbolic encodings of the sources of uncertainty in the evidence for the hypotheses. This knowledge is used by the control planner to identify and develop strategies for resolving the uncertainty in the interpretations. Since multiple, alternative strategies may be able to satisfy goals, the control process can be seen to involve a search. Heuristic focusing is applied in parallel with the planning process in order to select the strategies to pursue and control the search. The control plan framework allows the use of a flexible focusing scheme which can switch back and forth between strategies depending on the nature of the developing plans and changes in the domain.

6.3.1.3.1 Introduction

Interpretation is the process of determining a high-level, abstract view of a set of data based on a hierarchical specification of possible viewpoints. Hypotheses representing interpretations of some subset of the data are developed using the hierarchical support relations to identify the legal "evidence" for the hypotheses. Complex interpretation tasks require sophisticated evidential reasoning and control schemes which can deal with the uncertainty inherent in the interpretation process. For example, combinatorial considerations preclude complete construction and evaluation of all potential interpretations: control must be exercised over the creation and refinement of hypotheses. This means that the system must compare alternative hypotheses based on partial knowledge without being sure whether it has even created all of the correct interpretations. In many domains the uncertainty is compounded by the volume of data being too large to be completely considered and/or by the data being uncertain and possibly incorrect.

We have developed an interpretation framework based on a model of interpretation as a process of gathering evidence to manage uncertainty. The key components of the approach are a specialized evidential representation system and a control planner with heuristic focusing. The evidential representation scheme includes explicit, symbolic encodings of the *sources of uncertainty* in the evidence for the hypotheses. That is, evidence provides uncertain support for a hypothesis because there are conditions under which the evidence may fail to support the hypothesis: the sources of uncertainty in the evidence. For example, while some piece of sensor data in a vehicle monitoring system can be used to support a particular vehicle hypothesis, the resulting evidence is uncertain because the data may actually be due to a sensor malfunction or may support a competing, alternative vehicle hypothesis. Our model of interpretation associates a set of sources of uncertainty with the evidence for the hypotheses: partial evidence, uncertain evidence inference, uncertain evidence premise, alternative evidence interpretation (representing the relations between alternative hypotheses), conflicting evidence, etc. This knowledge is used by the control process in elucidating strategies for meeting the problem-solving goals since the *purpose* of interpretation actions is to resolve the uncertainty in the hypotheses. The evidential representation scheme is discussed in section 6.3.1.3.2.1.

Control decisions are made through an incremental planning process which identifies, selects, and implements problem-solving strategies. The available strategies are defined as a hierarchy of *control plans*. Control planning involves determining the subgoals of the current plan which must

be satisfied next and then matching each subgoal to the possible control plans to determine how it might be satisfied. Primitive control plans represent actions and have corresponding functions for executing the actions. Planning and execution are interleaved (i.e., the plans are only elaborated to the point of selecting the next action—because the outcome of actions is uncertain). In general, there will be many partial control plan instances which could be further elaborated at any point in the processing (i.e., many possible strategies for pursuing the system goals). The alternative control plan instances represent the choices of which hypotheses to resolve uncertainty in, what sources of uncertainty to eliminate, and how to do it. Thus, one of the major issues for interpretation systems is the development of an effective focus-of-attention scheme. In our system *focusing is accomplished as part of the multi-stage process of refining and elaborating the control* plans. Maintaining a framework of control plan instances allows the system a great deal of control flexibility since the focus-of-attention can move back and forth between strategies by refocusing in the plan instance hierarchy. Refocusing decisions are based on data-directed factors such as the outcome of control plans, the characteristics of the developing interpretation hypotheses, and data availability. Control planning and focusing are discussed in section 6.3.1.3.2.2.

The work presented here grew out of our experience developing a focusing scheme for plan recognition [17]. Plan recognition systems have tended to ignore the practical aspects of focusing the interpretation process and of comparing alternative hypotheses to determine just what it is that the system believes. The scheme we developed used an explicit record of the application of focusing heuristics to guide the system and allow it to revise its interpretations. However, the granularity of the control was too coarse for many domains, the focusing assumptions information was of limited value in controlling backtracking, and hypothesis uncertainty was confused with the control decisions (see [16]). Of interest to us also has been work on planning for control by Clancey [19], Hayes-Roth [42], and Durfee and Lesser [30]. However, none of these systems provide a completely suitable framework for interpretation. Clancey's tasks and meta-rules are really control plans and their substeps. The framework is limited by the fact that meta-rules directly invoke subtasks so there is no ability to search for and consider alternative strategies; focusing is implicit in the meta-rule preconditions. Of course, this may be fine for classification problems [20] which can be more exhaustive than can interpretation systems. The Hayes-Roth work on blackboard systems for control is more general than our work (which concentrates on interpretation). However, this generality means that little guidance is provided in how to structure control knowledge (e.g., the need for uncertainty knowledge as a basis for elaborating control plans). Another drawback of the control blackboard approach is its reliance on an agenda mechanism which must consider all possible actions on each loop. By contrast, focusing in parallel with the control plan hierarchy as we do provides, in effect, a partitioned agenda: only those actions immediately applicable to the in-focus plan or goal are considered. The incremental planning approach of Durfee and Lesser builds abstract models of the interpretation data and uses these models to guide further processing. This idea can be handled as one type of problem-solving strategy in our system with the addition of appropriate abstract operators and control plans representing the strategy. Most work on evidential representation systems relies on having a fixed set of alternatives among which to partition belief (as in classification problems) and having atomic hypotheses. As we discuss in section 6.3.1.3.2.1, this is not the case for interpretation so such work is not directly applicable. Our use of symbolic representations of uncertainty was inspired, in part, by Cohen's work on symbolic representations of evidence called *endorsements* [21]. These symbolic representation of uncertainty make it possible to understand the relations between the

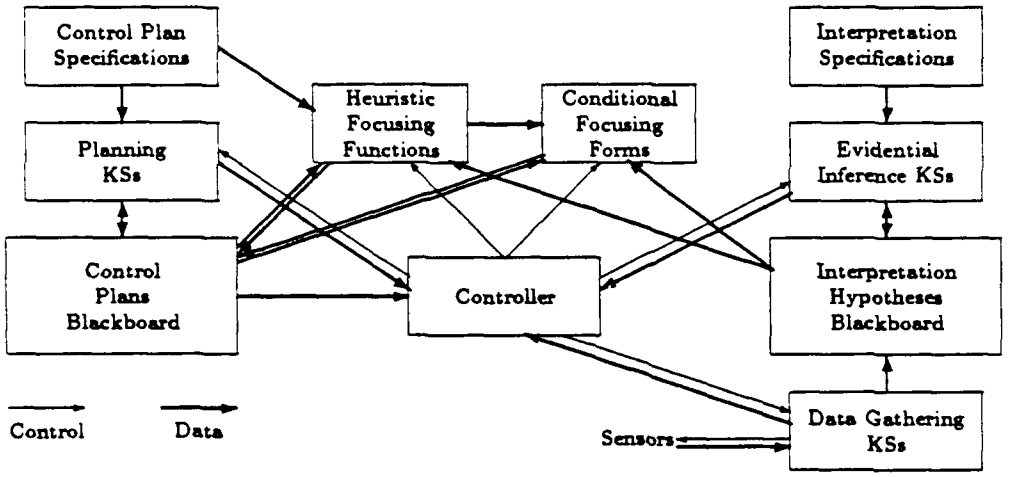

Figure 1: Architecture of System

hypotheses and the methods applicable to resolving the uncertainty. This is important because numeric summaries of evidential uncertainty do not provide the kind of information needed to understand how to go about resolving the uncertainty in the hypotheses.

6.3.1.3.2 Planning to Resolve Uncertainty

Figure 1 illustrates the major components of our system. The controller is responsible for executing the basic control loop (see Figure 4). Control plan instances created by the planning process are maintained on the Control Plans Blackboard. The expansion of the control plans is accomplished using Planning KSs which are created from the Control Plan Specifications. Interpretation hypotheses are developed by executing interpretation actions using the corresponding Evidential Inference KSs and the Data Gathering KSs. The hypotheses are maintained on the Interpretation Hypotheses Blackboard along with information about the evidence supporting the hypotheses, the uncertainty in that evidence, and the relations between the hypotheses. The Heuristic Focusing Functions are defined as part of the Control Plan Specifications and are applied to the control plans under the direction of the controller. The Conditional Focusing Forms are essentially demons which can be defined by the focusing heuristics in order to modify the flow of control and refocus within the control plan hierarchy. The satisfaction condition of each of these functions is checked following the execution of an inference or data gathering KS.

6.3.1.3.2.1 Evidence and Sources of Uncertainty

For each class of interpretation hypothesis, H, the interpretation hierarchy specification defines both a set of sets of lower-level, support hypotheses, $\{\{S_i\}_j\}$, and a set of higher-level, explanation hypotheses, $\{E_i\}$. Hypotheses may be supported by multiple sets of supporting hypotheses to model those domains in which there can be multiple *sources of evidence*. For example, in vehicle monitoring, vehicles might be supported by data from a variety of sensor types so that $\{S_i\}_1$ might represent the support from acoustic sensors and $\{S_i\}_2$ that from radar. The lowest-level units in the hierarchy correspond to the data to be interpreted. The highest-level units represent the most abstract interpretation of the data and require no explanation for their occurrence.

The model of evidence that we use is based on the requirements for control (i.e., that the *sources of uncertainty* for each hypothesis can be used to drive the control process). At any point during the interpretation process, each hypothesis instance is based on a *set* of evidential inferences of the form: $S_{k_l} \Rightarrow H$ (supporting evidence) or $E \Rightarrow H$ (explanation evidence) where $S_{k_l} \in \{S_i\}_l$, $\{S_i\}_l \in \{\{S_i\}_j\}$, and $E \in \{E_i\}$. There are then the following potential classes of sources of uncertainty for the correctness of a hypothesis:

- Incomplete Evidence:
  - The set of supporting evidential inferences $\{S_j\}_l$ (where $S_{k_l} \in \{S_j\}_l$) may be incomplete (i.e., $\{S_j\}_l \subset \{S_i\}_l$).
  - There may not yet be any explanation evidence.
- Uncertain Evidence:
  - The premise hypothesis of an evidential inference may be uncertain (i.e., some S_{k_l} or E_k is uncertain).
  - There may be alternative interpretations for the evidence—that is, for some $S_{k_l} in \{S_i\}_l$ the correct inference may be $S_{k_l} \Rightarrow H'$.
  - There may be alternative explanations for the hypothesis.
  - It may be uncertain whether an inference satisfies the hierarchy constraints (i.e., it is uncertain whether $\{S_j\}_l \subset \{S_i\}_l$ or whether $E \in \{E_i\}$ for H).
- Conflicting Evidence:
  - Support evidence doesn't exist (i.e., some S_{k_l} doesn't exist for H).
  - Explanation evidence doesn't exist (i.e., no $E \in \{E_i\}$ matches H).

As evidential inferences are made by the interpretation process, symbolic expressions are created which represent the specific instances of these classes of uncertainty which exist in the evidence. These symbolic expressions represent the sources of uncertainty in the evidence and are associated with the hypothesis the evidence supports. For example, the Attack hypothesis in Figure 2 includes the expression, (Alt-Interp "track" "recon"), to represent the uncertainty in its supporting track evidence due to the existence of an alternative interpretation for the track (the Recon hypothesis). Further examples of the symbolic sources of uncertainty may be seen in Figure 5 as the *result* of the plan Get-Sources-of-Uncertainty (which determines the sources of uncertainty which exist in the specified hypothesis). We have also used this framework to extend the knowledge typically found in an interpretation system by including sources of uncertainty for the evidential relations which do not result from alternative interpretations due to interpretation hypotheses. For example, acoustic sensor data in a vehicle monitoring system provides uncertain support for a vehicle because the data may be the result of a sensor malfunction or a weather disturbance. We can specify such factors as sources of uncertainty for particular evidence even though these factors are not explicitly included as interpretation hypotheses (there is no sensor malfunction hypothesis in the interpretation specifications). Because we represent these uncertainty factors, knowledge can be applied to confirm or disconfirm the uncertainty. However, when the methods for discounting these uncertainties require evidential reasoning and interpretation, then the factors must be included as additional interpretation hypotheses (e.g., ghost tracks).

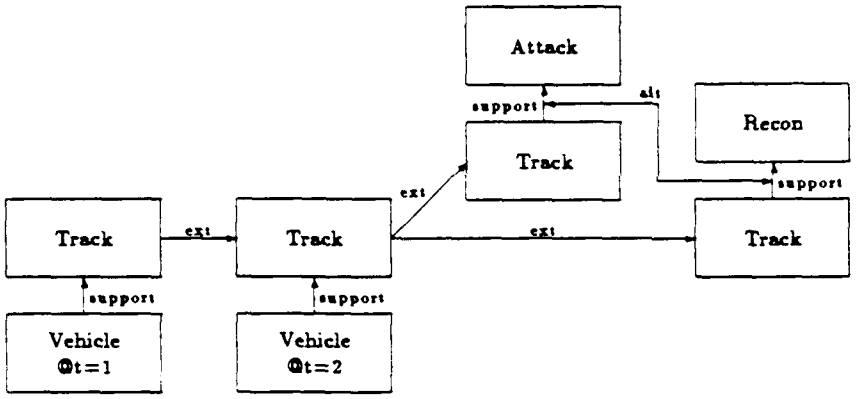

Figure 2: Hypothesis Representations

Interpretation hypotheses are compound structures because they may include parameters. The values of a hypothesis' parameters are defined in terms of the parameters of its support and explanation hypotheses. The inclusion of parameters (with continuous values) is one of the key characteristics of interpretation problems which distinguishes them from classification problems. Clancey [20] contrasts classification problem solving in which a solution is selected from a fixed set of alternatives with what he calls "constructive" problem solving in which a solution must be "formulated." He also points out that most medical "diagnosis" expert systems are really doing classification and are only usable for routine diagnosis in which the possibility of multiple, interacting diseases is ignored. Interpretation is clearly a form of constructive problem solving. We don't simply gather evidence to decide among a set of predetermined alternatives, but must gather evidence just to determine what the set of alternatives consists of. For example, in vehicle monitoring there is no way of knowing a priori how many vehicles the data might support nor where the vehicles might be. Sensor evidence for vehicle hypotheses not only supports the belief in a vehicle, it also defines the vehicle tracks by constraining the vehicle type and position.

Clancey mentions that the difference between classification and constructive problem solving has important consequences for choosing a knowledge representation. Gathering evidence for an interpretation hypothesis not only *justifies* the interpretation hypothesis, it also *refines* it by constraining its parameter values. Thus, every time evidence is added to a hypothesis it may result in a change in the parameter values of the hypothesis. However, since the evidence is uncertain, there may be alternative evidence which must also be pursued. Consequently, multiple versions of each hypothesis, called *extensions*, must be used to represent the alternative hypothesis refinements supported by the (uncertain) evidence. As part of our representation of the uncertainty in the hypotheses, we have developed a scheme for representing the alternative extensions of hypotheses and the interrelations between these extensions and between competing hypotheses. The advantage of this framework is that we can simultaneously reason about the uncertainty in a hypothesis due to alternative possible extensions of the hypothesis and due to competing hypotheses. Figure 2 shows a simple example of such an extension framework. In this example, there are four extensions of the Track hypothesis, each of which is caused by the addition of evidence which further constrains the Track. Of particular interest are the alternative extensions created by the competing interpretations of Track as support for an Attack mission or a Recon mission. These alternative extensions are used to recognize the *alternatives* relation between the support evidence for the Attack hypothesis and for the Recon hypothesis. The

Name	Eliminate-Sources-of-Uncertainty
Description	Eliminates the sources of uncertainty from the hypothesis ?hyp until the belief in ?hyp is greater than ?belief.
Goal Form	(Have-Eliminated-SOUs ?hyp ?belief)
In Variables	(?hyp ?belief)
Out Variables	()
Temp Variables	(?sou)
Subgoals	((Have-Source-of-Uncertainty . (Have-SOU ?hyp ?sou)) (Have-Eliminated-SOU . (Have-Eliminated ?hyp ?sou)))
Sequence	(ITERATION (GREATER (belief ?hyp) ?belief) (SEQUENCE Have-Source-of-Uncertainty Have-Eliminated-SOU))
Constraints	()

Figure 3: Control Plan Specification

representation framework also allows us to represent the relations between evidence at different levels in the evidential hierarchy. For instance, additional evidence gathered for one of the vehicle hypotheses may constrain that hypothesis in such a way that it is only consistent with either the Attack hypothesis or the Recon hypothesis—but not both. Since this evidence is uncertain, though, it cannot be assumed that it is correct. The representation creates alternative extensions of the relevant hypotheses in order to represent the relations created by the addition of this evidence.

The power of this representation comes from its usefulness for *differential diagnosis*. The basic strategy for resolving interpretation uncertainty is to gather support and explanation for the hypothesis (or else explain why the evidence cannot be found) and then do differential diagnosis to discount alternative uses of the data. Having complete support and explanation evidence for a hypothesis provides *necessary* evidence for the hypothesis, but it does not confirm the hypothesis because there still may be alternative explanations for all of the evidence. Thus the only way to gather *sufficient* evidence to confirm the hypothesis is to do differential diagnosis on the evidence. The explicit sources of uncertainty information in conjunction with the representation of hypothesis extensions provides the basis for doing this. In the example in Figure 2 the knowledge of the alternatives relation between the evidence for the Attack and Recon hypotheses allows us to understand why each is uncertain and what must be done to resolve the uncertainty. It also allows us to understand that each of these hypotheses negatively affects the belief in the other. Thus instead of trying to directly support a hypothesis we might also resolve its uncertainty indirectly by disproving its alternative.

Uncertainty in a plan hypothesis is typically resolved by gathering evidence to directly eliminate the sources of uncertainty for the hypothesis. However, in some domains it may also be possible to resolve uncertainty by gathering *independent* evidence for a hypothesis (i.e., using alternative sources of evidence). It should be noted, though, that evidence is not necessarily independent just because it is based on a separate source; independent evidence must include independent sources of uncertainty. For example, if evidence from radar and from radio emissions detection could both be affected by the same kind of weather disturbances then one source of evidence could not be used to resolve this source of uncertainty in evidence from the other source.

6.3.1.3.2.2 Control Plans and Heuristic Focusing

Control plans are defined using specifications like the one in Figure 3 for the plan Eliminate-Sources-of-Uncertainty. The goal of the plan is specified by the Goal Form, (Have-Eliminated

repeat: Pursue-Focus on each element of Current-Focus-Set until nil.

Pursue-Focus(focus)
case on type(focus):

plan	Focus on variable bindings to select plan instances for Current-Focus-Set.
	Expand plan instances to next subgoals.
	Focus on subgoals to select subgoals for Current-Focus-Set.
subgoal	Match goal to applicable plans.
	Focus on plans to select new focus elements for Current-Focus-Set.
primitive	Execute primitive plan action to get result.
	Update plan states and select new focus element for Current-Focus-Set.
	Check refocus and subgoal conditions.

Figure 4: Control Planning Loop

SOUs ?hyp ?belief). All of the variables in the goal form must be listed in either the Input Variables clause or the Output Variables clause of the specification. Input Variables must be supplied to the plan when it is instantiated and Output Variables are bound upon completion of the plan to return results. Many actions are capable of returning multiple bindings for variables (e.g., determining an available sensor unit). To deal with this fact, variables are allowed to take on *multiple valued* values which represent a set or range of bindings. Focusing then determines the value(s) to use in subsequent plan expansion. This example has the input variables ?hyp and ?belief in the goal form and no output or result variables. In addition to the input and output variables, the plan specification also includes a Temp Variables clause which lists variables used to hold any subgoal results that are not part of the plan goal form. Here, the temporary variable ?sou is used to hold the source of uncertainty being worked on by the plan. Each plan is realized by a sequence of subgoals. The subgoals of the plan are defined in the Subgoals clause in terms of the goal forms for the subgoals. These goal forms are used to identify control plans applicable to satisfying the subgoals by unifying the subgoal goal forms with the control plan goal forms of all possible control plans. In the example, there are two subgoals, Have-Source-of-Uncertainty and Have-Eliminated-SOU, whose goal forms are specified. The subgoal sequence is specified in the Sequence clause. This clause uses a shuffle grammar to express strict sequences, concurrency, alternatives, optional subsequences, and iterated subsequences. Eliminate-Sources-of-Uncertainty iterates a sequence of the two subgoals until the required level of belief in ?hyp is reached. The two subgoals represent the actions of identifying the current sources of uncertainty in ?hyp and eliminating a source of uncertainty in ?hyp. The plan thus proceeds by identifying the sources of uncertainty, selecting (through focusing) a source of uncertainty to be eliminated, eliminating the source of uncertainty, and repeating this sequence as necessary. The sequencing constraints of the Sequence clause can be augmented with additional ordering constraints and constraints on the subgoal variable values by placing additional conditions in the Constraints clause.

The basic control planning loop is detailed in Figure 4. An example control plan instance is represented in Figure 5 as an AND/OR tree of plan and subgoal nodes. The situation represented is such that the top-level plan to solve the interpretation problem has created a subgoal of resolving the uncertainty in the Attack hypothesis of Figure 2. In order to pursue this subgoal, the subgoal form is unified with the goal forms of the defined control plans to determine which plans are applicable to satisfying the subgoal. In this case, only a single plan is relevant to satisfying this goal, Eliminate-Sources-of-Uncertainty, the plan whose specification was shown in Figure 3. In general, there would be multiple matches corresponding to multiple possible

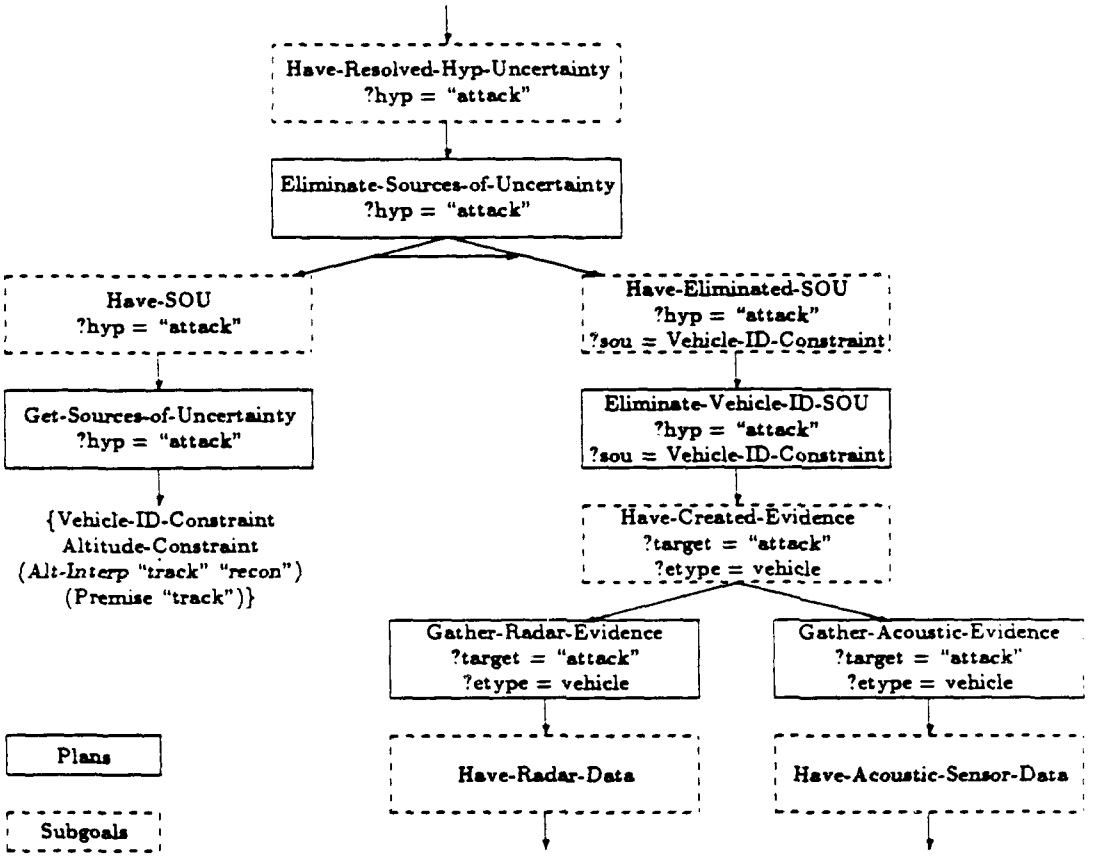

Figure 5: Control Plan Instance

strategies for satisfying the goal and focusing knowledge would be applied to select the plan(s) to focus on. Pursuing an in-focus control plan means expanding the control plan to create subgoals representing the substeps of the plan which need to be satisfied next. The first two subgoals of the Eliminate-Sources-of-Uncertainty plan are shown in Figure 5 even though they are sequential steps (signified by the horizontal arrow between their arcs). The first subgoal, Have-SOU, can be satisfied by the primitive plan, Get-Sources-of-Uncertainty. Primitive plans represent actions which may be carried out with corresponding Knowledge Sources. Primitives may generate information for the planning process (e.g., determining an available sensor to generate new data) or may generate interpretation evidence (e.g., to create an evidential inference). Actions may fail or may succeed and return results. In the case of Get-Sources-of-Uncertainty here, the action succeeds and returns a multiple-valued value consisting of the symbolic representations of the current sources of uncertainty in the attack hypothesis. This result is bound to the variable ?sou of the primitive and the status of the plan is set to complete. The outcome of the action (the change in status and the variable bindings) is propagated to the subgoal the primitive satisfies and then in turn to the plan containing the subgoal. The change in the state of the subgoal may mean that this plan has failed or has succeeded which would cause additional propagation. In the example, it simply changes the state of the plan so that it is expecting the next subgoal (Have-Eliminated-SOU) and binds the temp plan variable ?sou (which will be used to bind the subgoal

variable ?sou). Multiple-valued values such as that just bound to ?sou are used to represent a set of alternative bindings for a variable (i.e., uncertainty over the correct value). Because ?sou is used as an input to the next subgoal and because it has a multiple-valued value, it is necessary to apply focusing knowledge at this point to select the value(s) to be used for further expansion. The single-value version(s) of the subgoal are then used to match and select applicable control plans. In the example, the heuristic focusing knowledge in the Eliminate-Sources-of-Uncertainty control plan which is associated with the variable ?sou, has selected the source of uncertainty Vehicle-ID-Constraint as the (sole) in-focus binding for ?sou.

Focusing heuristics represent meta-level knowledge relative to the knowledge in the control plans. Whereas control plans embody problem-solving strategies for interpretation, focusing heuristics embody strategies for selecting the appropriate problem-solving strategies. In our framework, focusing heuristics are associated with particular control plans. There are several points at which focusing decisions must be made so we partition the focusing knowledge into four different classes: variable, subgoal, matching, and updating. *Variable* focusing knowledge is associated with each of the variables of the control plan environment and is used to select among competing bindings for a variable. This occurs when actions return *multiple-valued* values (as discussed above). *Subgoal* focusing knowledge is used to select among multiple active subgoals for a plan instance. Control plans can specify that certain subgoal sequences are able to be carried out in parallel, however, it still may be preferable to sequence the subgoals due to uncertainty over their satisfaction and results. *Matching* focusing knowledge is used to select among the multiple plans which are applicable to satisfying a subgoal when there are multiple plans goal forms which match a subgoal form. *Updating* focusing knowledge is associated with each subgoal of a control plan and is used to decide how to proceed when a plan for satisfying the subgoal completes (succeeds or fails). In general, plan expansion is not exhaustive so completion of a plan will leave alternative expansions of the plan and alternative matching plans which might be pursued to try to satisfy a subgoal. Updating heuristics decide whether to accept a result and propagate it or to pursue existing alternatives instead. Thus this knowledge is partially responsible for controlling "backtracking" of the system.

A common problem with meta-level focusing knowledge is conflicts. That is, we may have heuristic knowledge that says to "Prefer A" and other knowledge that says to "Prefer B" instead. The cause of such a conflict is the generality of the heuristics which fails to provide information about the proper context in which to apply the knowledge. For example, general heuristics for selecting the "best" data to use to create evidence might say to prefer data that is: well sensed, in time slices with a small number of data clusters, in tight clusters, etc. These heuristics may very well conflict by preferring different data. However, if we understand the purpose for which the evidence is to be used, we can avoid this problem because we understand that certain data characteristics are most important in particular contexts. Thus, the number of time clusters is less critical for extending a track than for creating evidence of a new track since the existing track constrains the data selection. Hierarchical control plans provide context with which to disambiguate heuristics since the control plan structure represents the purpose of the plans. Thus, by associating heuristic focusing knowledge with particular control plans and allowing the heuristics to examine the control plan hierarchy we provide the context to disambiguate the heuristics.

The basic control process described above is highly top-down and depth-first. However, the uncertainty of the interpretation process requires a strong bottom-up control component as well.

We accomplish this with several extensions to the basic focusing scenario which make it possible for the system to shift its focus between competing strategies in response to the characteristics of the developing plans and factors such as data availability. Focusing is extended by allowing variable and matching focus decisions to be: absolute, postponed, or preliminary. *Absolute* focusing heuristics simply select a single path to be pursued—subject, of course, to potential plan failure (which is handled by the updating process). However, focusing heuristics may not always be able to select a single "best" path to pursue. Instead, they may need to partially expand each of several competing strategies to gather more specific information about the situation before being able to select the best alternative.

We handle this nondeterminism by allowing multiple paths to be expanded with *postponed* focusing. In order to postpone focusing and pursue multiple paths, we must specify the conditions under which the alternatives should be reevaluated and how to reevaluate them. A postponed focusing decision creates a *refocus form* which specifies the paths to be pursued, the conditions for refocusing, and a refocus handler. Refocus conditions are evaluated following the execution of any action (only actions generate new knowledge). When they are satisfied, the refocus handler is invoked and reevaluates the choices within the new context in order to eliminate the multiple foci. An example of a postponed focusing decision occurs late in the refinement of the control plan in Figure 5. There are two control plans applicable to satisfying the subgoal Have-Created-Evidence: Gather-Radar-Evidence, which uses radar data to create the desired evidence, and Gather-Acoustic-Evidence, which uses acoustic sensor data. The system is uncertain about how to proceed because it cannot be sure which source of evidence will provide the best track evidence without knowing more about the actual data which is available. To handle this situation, the focus decision is postponed until the first subgoal of each alternative plan is satisfied (i.e., until the potential data is determined). The refocus handler is then used to evaluate the focusing alternatives in light of the additional information by evaluating the relative quality of the available data for each alternative plan.

Preliminary focus decisions are similar to postponed decisions except that refocusing involves a reexamination of all of the original alternatives as opposed to just those that were initially focused on. Preliminary focus decisions are used when one alternative is likely to be the best—subject to certain reservations about its progress or under a particular assumption about the situation. The refocus conditions can then monitor the progress of the choice or the validity of the assumptions. For example, they may be used to limit the amount of effort expended on one alternative by including refocus conditions which set a limit on the amount of time to be expended or the level of completion to be reached. This is important in plans like Eliminate-Sources-of-Uncertainty shown in Figures 3 and 5. Resolving the uncertainty in one particular hypothesis to the exclusion of all other alternatives could result in the system missing important domain activities or losing the opportunity to gather useful data. Thus such a plan would be selected with a refocus form which limits the amount of time spent on the plan or causes the choice of the plan to be reconsidered following each plan iteration.

Preliminary focus decisions may be combined with postponed decisions in order to pursue multiple options, refocus among them, and still limit the entire choice. They make it possible to define opportunistic methods for refocusing. Preliminary and postponed focus decisions also control the system's backtracking since they effectively define the backtrack points and the conditions under which the system backtracks. This provides the system with a form of nonchronological backtracking for a domain where dependency-directed backtracking is ineffective [16]. Toward

this end, the refocus conditions may also refer to plan failure.

6.3.1.3.3 Conclusion and Status

This work is a further example of the utility of making control decisions through planning. It expands on existing research with respect to planning for the control of an interpretation system in three significant ways: the control task is viewed as being driven by the need to resolve uncertainty, the uncertainty of the interpretation hypotheses is represented explicitly and symbolically, and the process of finding the correct control plan may be seen to involve a search process which requires focusing. The combination of control plans with parallel focusing and a symbolic representation of the interpretation uncertainty provides a flexible framework which can be used to implement sophisticated control strategies.

The use of control plans changes the nature of interpretation control reasoning. Typically, control decisions have involved first rating all of the interpretation hypotheses and the data to select the "best" item to pursue and *then* determining how to pursue it. Such decisions are extremely complicated since they have to simultaneously consider a variety of factors and they obscure the strategies being used since the strategies are implicit in the ratings functions. The fundamental problem with focusing on hypotheses is that it's really impossible to decide which hypotheses to pursue—let alone *how* to pursue them—without understanding *why* you are pursuing them (i.e., without knowing what purpose they serve with respect to the overall system goal). Rather than trying to include such knowledge (along with heuristic focusing knowledge) within a complex rating function, the control planning process described here expresses the problem-solving strategies as explicit control plans. The selection of hypotheses to pursue and the methods to pursue them then flows naturally from the selection of strategies to meet the current problem-solving goals and the instantiation of these strategies. In addition, hierarchical focusing in conjunction with plan refinement makes it practical to express the focusing heuristics explicitly as well. This is because only a limited number of alternatives are considered at any point and because the control plan structure provides detailed context information.

Currently, we have a prototype implementation of the interpretation framework presented here, which simulates a system for monitoring aircraft. A variety of data sources such as acoustic sensors, radar, and emissions detectors are included. Additional sources of evidence such as terrain, air defense positions, and weather information are also available. Active control over evidence gathering can be effected through control of the operations of some of the sensors. Interpretation hypotheses cover a variety of missions including those involving coordination of multiple aircraft. One of the areas of current research in this project is the issue of languages for expressing focusing knowledge. In particular, we are looking into the factors needed for focusing in real-time applications. This includes such things as estimated processing and elapsed times and estimates of the quality of the evidence from alternative strategies.

6.3.1.4 Planning with Worlds

A sequence of actions may be viewed as a series of state transitions. Existing context mechanisms (worlds) provide an efficient means for recording and inspecting this series of related states. Similarly, planning systems require a representation of the state at each point in an evolving plan. However, existing world mechanisms do not correctly capture the more complex semantics imposed by nonlinear, hierarchical plans. We examine the shortcomings of existing worlds systems when used for planning and present a modified world system design which overcomes these difficulties.

6.3.1.4.1 Introduction

The planning process involves the generation of a sequence of operators or actions to achieve a desired goal state starting from a given initial state. Planners must represent the plan as it evolves. The states in the plan comprise a set of related but distinct contexts in which queries can be made. Mechanisms such as truth maintenance systems (TMSs), assumption-based truth maintenance systems (ATMSs) and worlds systems provide a substantial basis for an efficient, powerful representation of such a series of related contexts.

From its inception, difficulties were recognized in using an ATMS to model time and action. In de Kleer's own words, "...problem solvers act, changing the world, and this cannot be modeled in a pure ATMS in which there is no way to prevent the inheritance of a fact into a daughter context."[51] Worlds systems were developed, in part, as "an approach to applying the ATMS to the task of representing the actions."[59] Further work has attempted to use the worlds mechanism in nonlinear, hierarchical planning [34, 53].

Our experience with using a worlds mechanism in such a planning system, however, has revealed that existing worlds systems fail to fully and correctly capture the semantics implicit in nonlinear, hierarchical plans. Briefly, there are three problems. The semantics are unacceptably affected by the order in which assertions are made. This results from an untenable assumption about "effectiveness" of actions [59]. Planners also require a richer set of network manipulation primitives than those present in the worlds mechanism. In particular, the planning process requires adding and deleting inheritance links and creating and deleting worlds. Finally, the query mechanism in the worlds system is based only upon ancestor worlds while planners must also consider worlds that are not now, but may later be ancestors. Here, our goal is to describe these shortcomings and present a modified world system design which overcomes these difficulties.

In the following section, we describe the requirements imposed upon worlds systems in order to support planning. An extended query mechanism for nonlinear planning is developed in Section 6.3.1.4.3. Section 6.3.1.4.4 explains the extensions necessary to correctly support plan network modifications.

6.3.1.4.2 Using Worlds for Planning

In general, nonlinear hierarchical planners represent plans as directed acyclic lattices of *plan nodes*, with partial temporal orderings among the nodes. The nodes contained in these *plan networks* represent both primitive *actions* and more abstract *goals*. To refine a plan, the planner expands the plan network by either *phantomizing* or *expanding* the goal nodes. A goal node may be phantomized by either recognizing that its goal is already satisfied at its current point in the

plan or by imposing additional orderings between nodes in order to assure the goal's satisfaction. If the goal cannot be phantomized, an *activity* is selected to accomplish the goal and a plan network representing that activity replaces the goal node in the original plan.

To perform this style of planning, a planner must be able to determine if a (goal) fact is true at a particular point in a plan and, if not, what additional constraints could be imposed on the plan to assure that it is true. We call this part of the planner a query mechanism. The planner must be able to transform the plan so that it satisfies additional constraints. These additional constraints may be discovered by the query mechanism or derived from domain knowledge and heuristics about it. The heuristics and domain knowledge used by a planner are crucial, but beyond the scope of our discussion here. There are three types of constraints which the plan network must accommodate. Temporal ordering or parts of the plan is the first constraint type. Asserting that a fact is true (or false) at some point in the plan is the second type. Binding a variable to another variable or a domain object is the final constraint type. Since hierarchical planners search a space of partial plans it is also important to be able to undo any of these transformations.

Thus, in order create and maintain a plan network [7], one must be able to:

1. Generate a partially ordered, acyclic plan network;
2. Inquire whether a fact is necessarily true as some point in a plan or whether it can me made true by imposing additional constraints;
3. Expand any action in the plan into a network of actions;
4. Add arbitrary additional orderings between actions (such that they do not violate any existing orderings); and
5. Bind variables to other variables or domain objects.

Existing worlds systems are almost, but not quite, ideally suited for such a task. In this section we show how current worlds systems fail to correctly capture the semantics of these nonlinear plans and how their functionality must be augmented to support this type of planning.

To represent the state of the domain at each point in a plan, a world hierarchy may be constructed corresponding to the developing plan network. Thus, for each node in a plan network there exists a corresponding world. Temporal "successor" links between nodes in the plan network are mirrored by "descendant" links between the corresponding worlds in the world hierarchy.

As we will see, the functionality of current worlds systems must be modified and augmented to provide this support. In particular, existing query mechanisms must be modified and network modification facilities must be extended. In the following sections, we examine the semantics required by plans more closely and present modified query and network modification mechanisms in detail.

6.3.1.4.3 The Semantics of Queries

In order to understand the semantics imposed upon a worlds system by a nonlinear hierarchical planner, we must first define the semantics of such a plan. We begin by defining a plan as a linear set of *plan states* linked by *actions*. We then extend this to allow variable references within the actions. Finally, we further extend the definition to include nonlinear plan networks.

6.3.1.4.3.1 Queries in Linear Plans

Our initial definition of a linear plan closely follows that of [65]. First, we define:

1. $\mathcal{C}$ is a set of all possible wffs;
2. $\mathcal{W}$ is the set of all possible world states, (i.e., all possible subsets of $\mathcal{C}$);
3. $\mathcal{I}$ is a set of initial states such that $\mathcal{I} \in \mathcal{C}$;
4. $\mathcal{G}$ is a set of goal states such that $\mathcal{G} \in \mathcal{C}$;
5. $\mathcal{O}$ is a set of operators (or actions) where $o_n \in \mathcal{O}$ and $o_n \equiv < P, D, A >$ where P is the set of preconditions; D is the set of deleted wffs; A is the set of added wffs; and P, D and A are each subsets of $\mathcal{C}$;
6. For each operator o_n, we define an $\hat{o}_n$ such that:
$$\hat{o}_n(s) \equiv \begin{cases} (S \backslash \mathrm{D}) \cup \mathrm{A} & \text{if P} \in S \\ \perp & \text{otherwise} \end{cases}$$

Thus, a linear plan may be defined as the vector $o_1 \cdots o_n$. We evaluate a fact f for a plan $o_1 \cdots o_n$ using a query function Q where:
$Q(o_1 \cdots o_n, f) \equiv f \in \hat{o}_1 \circ \cdots \circ \hat{o}_n(s)$ for all $s \in \mathcal{I}$.

6.3.1.4.3.2 Queries in Linear Plans with Variables

We next consider the case where variables are contained in either the query or the wffs. A plan now requires both the previously defined operator vector as well as an interpretation function to represent bindings on the variables. The query mechanism must therefore take the current interpretation into account and return the set of all minimal extensions of this interpretation under which the queried fact is true. If the set of returned extensions is empty, the queried fact is false; if the set contains exactly the current interpretation, the fact is true; otherwise, the fact is only true when the current interpretation function is extended to include any of the returned interpretations.

6.3.1.4.3.3 Realistic Queries in Nonlinear Plans

The linear query mechanism can be extended to nonlinear plans by considering all linearizations of a partially ordered plan network. In addition to the extended interpretation function described above, the query must now also return the set of acceptable linearizations. If the queried fact is true for all possible linearizations and the returned set of interpretations contains exactly exactly the current interpretation, the fact is true; if there is no acceptable linearizations or if all of the acceptable ones contain no possible interpretations, the fact is false; otherwise, the fact is only true when one of the returned linearizations and interpretations are chosen.

Traditional world systems do not consider interactions between parallel worlds[2] until the point in the hierarchy, if any, at which they merge. Because planning systems can impose an

[2]We will refer to worlds as "parallel" if there is no ordering between them in a given world hierarchy. This does not imply that they must remain unordered, only that such an ordering is not currently specified.

ordering between currently parallel worlds, such interactions must be considered. Thus, queries about the state of a world must consider the possible effects of assertions and deletions in parallel worlds. In this section we introduce a *realistic query* for nonlinear plans.

As pointed out in [59], the "truth" of a fact in a merge world can be ambiguous and may be interpreted in several different ways. In an *optimistic* merge, a fact is true if there are any linearizations of the parallel worlds in which the fact is true. Conversely, in a *pessimistic* merge, a fact is true only if it is true in all linearizations. For planning systems, the potential for ambiguity exists not only at merge worlds, but at any world that is parallel to some other world. *Realistic queries* account for these potential interactions with parallel worlds and return the conditions under which a fact f true in a world w.

Consider first an optimistic query of a fact f' in world w. The function $q_o(f', w)$ returns the set of worlds which caused f' to be true at w. That is, f' was asserted at each of these worlds, w', and was not added or deleted at any world between w' and w.
Formally, the optimistic query is defined:

$$\begin{aligned} q_o(f', w) \equiv \{w' \mid & \neg\text{after}(w', w) \wedge \text{Add}(f', w') \wedge \\ & \neg\exists w'' \mid (\underline{\text{before}}(w'', w) \wedge \underline{\text{before}}(w', w'') \wedge \\ & \qquad \text{Add}(f', w'') \vee \text{Delete}(f', w''))\} \end{aligned}$$

where "before" and "after" indicate strict orderings and where $\text{Add}(f, w)$ indicates that the fact f was asserted in world w and $\text{Delete}(f, w)$ indicates that the fact f was deleted in world w.

Furthermore, the fact being queried may contain variables. An asserted fact need not be exactly equal to the queried fact in order to make it (possibly) true. We must therefore consider the assertion (and deletion) of facts which unify with the queried fact.

A query can be satisfied by finding a fact which unifies with the queried fact and is believed to be true in the world in which the query is made. More formally, $q_{\mathcal{U}}(f, w)$ returns the set of pairs of facts and worlds $(< f', w' >)$ such that f', which was asserted in world w', unifies with f, and f' is optimistically true in w.

$$q_{\mathcal{U}}(f, w) \equiv \{< f', w' > \mid \mathcal{U}(f, f') \wedge w' \in q_o(f', w)\}$$

This, however, is too optimistic. The function d computes the set of worlds at which deleted facts might make the query false (i.e., the "deniers" of the queried fact). We identify these worlds to allow the planner to prevent these worlds from interfering. This can be done either by "linking out" these worlds, or by constraining variable bindings so that the asserted facts, the deleted facts and the queried fact do not all unify.

More formally, given a query of fact f at world w and a fact f' which unifies with f and is added at world w', the function d returns the set of worlds that may be between w and w' and at which f'' (which unifies with both f and f') is deleted, thereby possibly making f false at w.[3]

$$\begin{aligned} d(f, w, f', w') \equiv \{< f'', w'' > \mid & \text{Delete}(f'', w'') \wedge \mathcal{U}(f, f', f'') \wedge \\ & \neg\text{before}(w'', w') \wedge \neg\text{after}(w'', w)\} \end{aligned}$$

[3]Even if a fact is queried at the same world in which it is made true, there may be a parallel world at which it is denied. If this semantics is not what is desired, (i.e., if a fact should always be considered true in a world at which it is asserted, then a clause requiring that w and w' be different can be added).

Finally, the *realistic query* of fact f at world w returns tuples containing assertions which could make f true at w and sets of worlds which potentially defeat those assertions.

$$q_r(f,w) \equiv \{< f', w', d(f,w,f',w') > | < f', w' > \in q_{\mathcal{U}}(f,w)\}$$

The results of a query to q_r may be interpreted as follows:

1. If no tuples are returned by $q_r(f,w)$, there is no known way to make f true in world w.
2. f is known to be true in world w if and only if the returned set of tuples contains at least one element such that:
   (a) the third element (i.e., the set of deniers) is empty,
   (b) w' is strictly before w, and
   (c) $f = f'$
3. If neither of the above conditions hold, each tuple represents a way in which f can be made to be true in w. to accomplish this, w' must be made strictly before w and the following must be done for at least one tuple:
   (a) variables in f' must be bound so that $f' = f$, and
   (b) for all deniers in the tuple, either:
      i. variables in f'' must be bound such that $\neg\mathcal{U}(f', f'')$, or
      ii. orderings must be imposed between worlds such that w'' occurs either strictly after w or strictly before w'.

Note that for a tuple where the deniers cannot be removed (i.e., adding the required world orderings is not possible and no satisfactory bindings can be made) the tuple offers no support for the fact and may be deleted.

6.3.1.4.4 Modifying the Plan Network

Plan networks are modified by changing world states and by changing the structure of the plan. The order in which these changes are made should not affect the semantics of the resulting plan. That is, given an initial state of the world hierarchy and a set of transformations, a *realistic query* on the world state resulting from *any ordering of these transformations* produces the same result. However, the semantics of existing worlds systems are affected by the order in which both of these types of transformations are made. The following section explains how the *realistic query* is implemented so that it is independent of the order in which adds and deletes are made. Section 6.3.1.4.4.2 explains how the network modification primitives are made order independent.

6.3.1.4.4.1 Dependence on Assertion Order

In existing worlds systems, the *effectiveness assumption* prevents assertions and deletions from being recorded unless they change the state of the world in which they are made. This results in queries being sensitive to the ordering in which the assertions and deletions are performed. No

such assumption is made in our system and the resulting query mechanism is therefore completely independent of the ordering of the transformations. This is accomplished by maintaining lists of worlds in which a fact is asserted. When a deletion is performed, the list of assertions for that fact is searched and ineffective assertions are modified to obtain the correct semantics. An analogous modification is needed to correct the semantics for deletions performed before additions of the same fact.

6.3.1.4.4.2 Modifying the Plan Structure

Implicit in existing worlds systems is the assumption that the structure of a world hierarchy will only be modified by the addition of new leaf worlds. This assumption of monotonicity in existing worlds, as embodied by the *effectiveness assumption*, is not valid when using the world hierarchy to represent a plan. First, ordering links may be added between existing worlds, corresponding to the addition of temporal orderings of plan nodes. Second, new worlds may be created which have existing worlds as children, such as during the replacement of an existing goal node with its expansion.

Consider the case in which a fact is asserted in one world and deleted in a parallel second world. The deletion is represented by causing the assumption that the fact is asserted to be inconsistent with the world in which it is deleted. The ATMS representation of such an inconsistency is called a "nogood" set. If the worlds become ordered as a result of the planning process such that the "deleting" world precedes the "asserting" one, the existing set of nogoods would make the fact appear false in those worlds below and including the original join world. To compensate for this, certain nogoods can be removed. This set of nogoods is determined as follows.

Consider a world hierarchy containing two nodes, w_h and w_t (for head and tail) which are not currently ordered but are to be linked such w_t is made a child of w_h. We first define the maximal common ancestors of these two nodes, $\text{mca}(w_h, w_t)$, to be the set of worlds, w_a, before both w_h and w_t such that there is no intervening world, w_i, which comes strictly after w_a but before both w_h and w_t.

$$\begin{aligned}\text{mca}(w_h, w_t) \;=\; & \{w_a \mid \text{before}(w_a, w_h) \wedge \text{before}(w_a, w_t) \wedge \\ & \neg(\exists w_i) \mid (\text{before}(w_i, w_h) \wedge \text{before}(w_i, w_t) \wedge \\ & \qquad \underline{\text{before}}(w_a, w_i))\}\end{aligned}$$

Similarly, minimal common descendants of the nodes, $\text{mcd}(w_h, w_t)$, are the worlds, w_d, which both w_h and w_t are before such that there is no intervening world, w_i, which comes strictly before w_d but both w_h and w_t are before.

$$\begin{aligned}\text{mcd}(w_h, w_t) \;=\; & \{w_d \mid \text{before}(w_h, w_d) \wedge \text{before}(w_t, w_d) \wedge \\ & \neg(\exists w_i) \mid (\text{before}(w_h, w_i) \wedge \text{before}(w_t, w_i) \wedge \\ & \qquad \underline{\text{before}}(w_i, w_d))\}\end{aligned}$$

Next we need to find the worlds in which assertions and deletions could possibly be affected. For asserts, we find worlds, $w^+(w_h, w_t)$, which come before w_h and which is strictly preceded by at least one maximal common ancestor. For deletions, the worlds, $w^-(w_h, w_t)$, must be preceded by w_t and must strictly precede at least one minimal common descendant.

$$w^+(w_h, w_t) \;=\; \{w \mid (\exists w_a \in \text{mca}(w_h, w_t)) \mid$$

$$\underline{\text{before}}(w_a, w) \wedge \text{before}(w, w_h)\}$$

$$w^-(w_h, w_t) = \{w \mid (\exists w_d \in \text{mcd}(w_h, w_t)) \mid \underline{\text{before}}(w, w_d) \wedge \text{before}(w_t, w)\}$$

Finally, the set of "nogoods", $\text{ng}(w_h, w_t)$, may be expressed as triples $\langle f, w', w'' \rangle$ such that f is a fact, w' is a world contained in w^+ in which f is asserted, w'' is a world contained in w^- in which f is deleted, and there is no known ordering between w' and w''.

$$\text{ng}(w_h, w_t) = \{\langle f, w', w'' \rangle \mid w' \in w^+(w_h, w_t) \wedge w'' \in w^-(w_h, w_t) \wedge \text{Add}(f, w') \wedge \text{Delete}(f, w'') \wedge \text{parallel}(w', w'')\}$$

6.3.1.4.5 Conclusions

We have shown that a worlds mechanism, as previously described, is not powerful enough to serve as a plan representation for hierarchical nonlinear planning. Starting from a detailed analysis of these problems, we have derived a modified world semantics to support planning. We have implemented a worlds system incorporating these extensions and this system has been used to build a working planner demonstrating that these extensions are sufficient.

6.3.1.5 Plan Execution Using Human Agents

Most planning systems have been applied to simple domains. In complex domains, the autonomy of human agents and the dynamic nature of realistic settings give rise to frequent *exceptional occurrences (exceptions)*. Rather than using a traditional error recovery approach, we advocate the use of plan recognition techniques to identify the purposeful behavior underlying an exception and its contribution to an ongoing plan. We discuss a model of plan execution and exception handling, and describe SPANDEX, an implementation of this approach. The SPANDEX system produces explanations consisting of *rationales* and *amendments* to incorporate exceptions into the current plan, allowing planning and execution to continue.

6.3.1.5.1 Motivation

Planners can potentially be used to automate or support a variety of complex tasks [53, 70, 84]. Most planning research, however, has been done in very simple domains (e.g., the blocks world) The dynamic and unpredictable nature of many real world domains suggests that sophisticated monitoring of plan execution is vital and systems should have the capability to respond to unexpected change. Recovery measures such as those of [1, 41, 84] have been proposed to effectively replan around an unanticipated domain state change, allowing resumption of the task while preserving as much of the plan as possible.

In our work, we are concerned with the special requirements of domains where a planner is used to support the cooperative work of one or more human agents [53]. In such environments, human input is required to guide the development of a plan for a task. In addition, execution of plan steps will be performed by human agents as well as by the planning system. The natural intelligence and familiarity of humans with the application domain means that their actions, even when inconsistent with system expectations, are generally purposeful. That is, human-generated plan exceptions should be incorporated into the developing plan, rather than "undone" using replanning techniques.

A planner employs a set of axioms that defines the planning process and a predefined set of domain activities and objects to generate valid plans that accomplish a particular goal. While restricting a potentially explosive search space, the plans that are produced are stereotypical and may not be adequate predictors of subsequent execution behavior. Our approach is to make a conventionally produced plan "elastic" in response to exceptions and to thus allow the continuation of planning and execution [12, 9, 11]. The domain knowledge base is used in an attempt to transform the current plan into a valid alternative, or, put another way, to recognize an alternate plan. During this process, additional domain knowledge may be acquired. The overall goal of our approach is to allow the system to continue planning and execution while incorporating the exceptions that occur in real domains.

Here we give a detailed formulation of plan execution and the exception problem. We then describe iterative and interactive algorithms which provide explanations for exceptions by establishing plausible rationales and proposing corrective measures. In the final section, we describe the implementation of these ideas in the SPANDEX exception handling system using an example from the software development domain.

6.3.1.5.2 Planning

In this section, we give a formalization of the planning problem and define terminology which is relevant to plan execution and exception handling. Our definition of the planning task of a hierarchical nonlinear planner is similar to that proposed in [65]:

> Given:
>
> - w: an initial world state
> - A. a set of activities, some of which are primitive (A_p) and others which are complex (A_c);
> - g: a desired final goal state;
> - E: a set of available agents;
>
> Determine: a partial ordering P of primitive activities A_p in A which, when executed in an initial state w by agents in E, will produce a new state containing the final goal state g.

Activities are considered complex if they can be elaborated by the planner into sub-activities; primitive activities are associated with executable actions. The partial ordering P which represents the final plan for a task is the result of a series of transformations of the initial goal specification g. The result of each of these manipulations is represented by a *plan network*, which represents the current version of an evolving plan. A plan network is a strict partial order and consists of the following elements[4]:

- N: a set of nodes, where each node represents a *goal* or *activity*;
- L: a set of temporal links which establish a partial ordering among the nodes;
- W: a set of world states, which are snapshots of the dynamic domain knowledge base. Two of these world states are attached to each node in N to describe the world states believed by the planner to hold before *(before-world)* and after *(after-world)* the execution of that node;
- I: a set of *protection intervals*, where each interval specification designates a partial world state and the temporal range during which it must be maintained.

Plan networks can also be described in terms of the stage of their execution. Since the domains we are concerned with generally interleave planning with execution, plan networks are often partially executed. Associated with each plan network is a set of *expected action nodes* which are the nodes which can be executed next. A node is in this set if and only if[5]:

- it is a primitive activity node;
- all of the conditions specified in the node's before-world are satisfied;
- all of the node's necessary predecessors are complete and awaiting successors.

[4] More detailed descriptions of each of these elements can be found in [53].

[5] A more complete definition of "ready nodes" which defines "conditions" and "necessary predecessors" in detail can be found in [53].

The process of *planning* is viewed as iterative *transformations* on plan networks. A complete *plan* is a plan network which has been fully ordered, and every node is either a *phantom*[6] or it is a primitive activity node that has already been executed. Thus, a plan network represents a class of complete plans; there are multiple possible complete plans that may result depending on the choices of elaborations and operations that are subsequently applied. As a plan network is further elaborated and executed by the plan network maintenance system (PNMS[53]), a *plan history* is built up since a new plan network results each time a PNMS operation is performed. PNMS operations include node expansions, the imposition of temporal orderings and protection intervals, etc. The complete *plan history* is the set of all intermediate plan networks created by the planner. Thus, the plan history is a partial order of plan networks ordered within planning time, where the distinguished upper bound is the eventual complete plan. The relation is a partial order since backtracking may be allowed. The plan history maintains a record of all planning actions performed in the production of the final plan.

We define the concept of a *plan wedge*[7] in order to be able to refer to the portion of plan history that represents the abstractions and subsequent refinements which introduced a given node n into the plan. The concept of a wedge is important both for general replanning and in establishing a rationale for how an unanticipated event may be relevant to the plan history. A *plan wedge* for a node *n* is a set of nodes defined as follows:

> Given the following recursive functions:
>
> 1. **node-ancestors(n)** which returns set of nodes containing the parent node which produced the expansion containing **n** as well as all **node-ancestors(parent node)**, and
> 2. **node-descendants(n)** which returns all children nodes which form the expansion of **n**, as well as all **node-descendants(child)** for each of the children nodes,
>
> a *plan wedge* consists of a distinguished node in **node-ancestors(n)** which is chosen as the *apex* of the wedge, and the set of nodes in **node-descendants(apex)**.

6.3.1.5.3 Plan Execution and Exceptions

We can now describe how exceptions arise during the planning and execution process. We sketch the system loop in order to provide an overall context:

1. The planner completes an elaboration cycle of the current plan network.
2. An action is executed and incorporated into the plan network.
3. Inconsistencies in the plan network resulting from the executed action are calculated.
4. *Rationales* are generated as justifications for the inconsistencies, along with proposed *amendments* that will restore a consistent system state.

[6] A phantom node [81] is a goal node which has been determined to be true at its position in the plan without further expansion and execution.

[7] Our definition is similar to the definition of a wedge used by Wilkins [84] and produces the semantic equivalent.

5. A *rationale* and an *amendment* are chosen through an interactive dialogue with the user. If no explanations are produced or considered acceptable, the exception may represent a user error. SPANDEX is also capable of interpreting a limited set of common user errors, which are based on models of procedural error types or "slips" [43, 67, 68].

6. Any inconsistencies which might remain are handled by standard replanning techniques.

7. Planning and execution resume (new elaboration cycle).

In the remainder of this chapter we describe in detail the detection and explanation of exceptions.

6.3.1.5.3.1 Detection of Exceptions

Given a plan network p with an identified set of expected action nodes, an action a may be executed with the resulting world state w. The execution event is denoted by (a, w). The activity descriptor a consists of an *operator* and *parameters*. The operator name is assumed to uniquely identify the activity and the parameters refer to domain knowledge base objects which are being manipulated by this activity. For example, a might be *compile-file(module-1)*. The operator in this case is *compile-file*, where the file being compiled by this activity is *module-1*.

If the specification of a unifies with one of the expected action nodes, that expected action node is processed accordingly to reflect that it has been executed, and no further changes are made at this time to the plan network. Otherwise, a node representing a is inserted into the network at the current point in execution time, so that it occurs after all executed nodes and prior to any expected action node, and resulting inconsistencies are calculated.

Therefore, the problem now posed to the SPANDEX exception handling system can be stated as follows:

Given:

- p: a partially executed plan network;
- (a,w): an event-result token;
- I: a set of calculated inconsistencies.

Produce a new successor plan-network p' which meets the following criteria:

- The set of executed nodes in p' include all executed nodes in p;
- p' contains a node representing the exceptional action;
- p' contains no inconsistencies.
- p' has the same high-level goal as p.

Our general approach to this problem is to manipulate available domain knowledge to generate plausible *explanations* which indicate how the current network and domain knowledge base can be transformed to eliminate inconsistencies resulting from the occurrence of (a, w). Inconsistencies must be one or more of the following:

1. The action type of a doesn't match with the types of any of the *expected action nodes*.

2. The action type of a matches with the type of one of the *expected action nodes*, but the parameters of a and the parameters of the expected action node do not match.

3. As a result of changes reflected in the new world state w, the plan network may now be inconsistent. In other words, a violation may be detected of one or more of the *plan network consistency criteria* defined below:

   (a) All before-worlds and after-worlds in W are internally consistent with respect to domain constraints.

   (b) The preconditions of each plan step are satisfied in its before-world.

   (c) The after-world of each plan step must be consistent with the goal of the plan step.

   (d) All protection intervals in I must hold.

   (e) The set of temporal ordering specifications L must be consistent.

The procedure followed up to this point (exception detection, insertion of a node representing the exception, and subsequent problem computation) is very similar to that followed by SIPE's exception handling component [84]. SIPE and other systems [1, 41] have also categorized potential plan "flaws"' that may be introduced as a result of an unexpected state change. These flaws can be shown to be a subset of the above categorization of inconsistencies. In SIPE, all exceptions are treated as "mother nature" occurrences, handled by simple insertion into the plan network followed by generic recovery actions. Neither SIPE nor other replanning systems make any attempt to establish any correlation between an unexpected event and other elements of the ongoing plan, whereas the remainder of the SPANDEX task is to do exactly that. In the next section, we discuss how explanations are constructed to justify exceptions and eliminate inconsistencies.

6.3.1.5.3.2 Explanation generation

In the previous section we have enumerated the types of inconsistencies that can result from an unexpected user action. Explanations for these inconsistencies are generated by the controlled application of a set of ***plausible inference rules*** (PIs). Each PI maps from an inconsistent state specification S to an explanation E. S consists of a set of identified inconsistencies which are constrained by one or more specifications of relations between domain knowledge base objects. For example, the inconsistent state specification S of the PI which is used in the example in Section 6.3.1.5.4 is the following: "If the inconsistency is unexpected-action-type(a), and specialization-of(action-type(a), action-type(expected-action)), then ..." The explanation E also has two components: a *rationale*, and an *amendment*. The rationale gives a semantic basis for the exception, suggesting its contribution to the ongoing plan. Examples of rationales are:

- This is an alternative way of performing an expected action.
- This is an alternative way of accomplishing an abstract goal or activity which is *in-progress*, which means that one or more of the subnodes of the abstract node has been executed.
- This is an alternative action which represents a shortcut in the plan (some steps may be skipped).

- Actions are being performed out of order and ordering may be relaxed.
- This is a new action which was not known to be part of the plan and should be added to the static task description.

An amendment prescribes the changes to be made in order to establish the rationale and restore system consistency. It consists of one or more of: a predefined set of *plan network alterations*, and primitive modifications on the domain knowledge base. The plan network alterations are composed of the primitive plan network operations *delete-node*, *insert-node*, *expand-node*, and *establish-ordering*. Examples of plan network alterations include: 1) replacing one of the expected actions with a node representing the unexpected action, 2) replacing a wedge containing one of the expected actions with a node representing the unexpected action, and 3) replacing a later activity node with a node representing the unexpected action and deleting the intervening nodes. The modifications that may be made to the domain knowledge base include the addition or deletion of values to a field of an object, adding or deleting a taxonomic link, or modifying a constraint.

The most likely explanations will be generated by the application of PI rules whose inconsistent state specification S holds completely in the current world model, and are referred to as *complete* explanations [9]. However, since we are interested in adding to an inherently incomplete domain model, we want also to consider rules whose inconsistent state criteria are not entirely met; we attempt to establish the missing information through interactive dialogue, thus producing additional plausible explanations while adding to domain knowledge. In order to intelligently control the application of PI's, we use a set of heuristics similar to those applied to plan recognition problems [17]:

- **Completeness**: Prefer a plausible inference rule which has more components in its inconsistent state specification S that are true in the world model to a rule with fewer true components.
- **Locality**: Prefer a plausible inference rule that considers an expected action (or wedge) to one considering a later action (or wedge).
- **Cost**: Prefer a plausible inference rule that proposes fewer modifications in its amendment to one proposing more modifications.

A threshold is set to limit the number of explanations produced. These most likely explanations are presented to the user in an interactive fashion and a choice is requested. If none of these explanations are acceptable, the process is iterated and the next set of explanations are produced and presented, until an explanation is selected. If no explanation is selected, SPANDEX attempts to fit the exception into one of its known common error classes. If an explanation is selected, the amendments are applied, and SPANDEX must check the resulting network for consistency. Any remaining violations are handled through standard replanning techniques or through an interactive acquisition session with a human agent.

6.3.1.5.4 Example

In this section we present an example from the domain of software engineering, one of the domains which is currently implemented in SPANDEX.

The overall goal of the example task is to create a new version of a software system, incorporating desired changes and additions. A partial plan network is generated for this task, and is executed in conjunction with the relevant agents. Three ordered subgoals are generated for this task: (*decide-on-changes* (the programmer must decide which particular changes to make), *make-changes* (the editing must be performed on the appropriate modules)) are expanded and accomplished, and *have-consistent-system* (the entire software system must be updated so that changed modules are recompiled and the system is relinked).

After expanding and accomplishing the first two subgoals, the planner attempts to achieve the third subgoal *have-consistent-system* by selecting the activity *update-software-system*. Upon requesting verification from the user to perform the first primitive action in this activity expansion (*compile* the first changed file), the user denies verification and instead initiates a *unix-make* action. SPANDEX determines that an action mismatch has occurred, implying a possible attempt at an action substitution or an out-of-order action[8]. An *exception record* (see Figure 1) is created to summarize the exception. The exception analyst module of SPANDEX then uses a heuristic *rationale.selector* to choose a method to generate rationale records for the exception.

Unit-name: ACTION.TYPE.MISMATCH.01
Unit-comment: "The type of action performed did not match an expected action type."
Exceptional-action: unix-make-01
Exception-summary:
"The target action: *compile-file-01* did not occur;
unix-make-01 was performed."
Target.action:*compile-file-01*
Perceived.action *unix-make-01*
Rationales: alternative.action.rationale, out.of.order.action.rationale
Rationale.selector: rationale.selector.method
Rationale.records: alternative.action.rationale.01

Figure 1: Exception record (ACTION.TYPE.MISMATCH.01)

In this example, a single applicable plausible inference rule is retrieved, and SPANDEX constructs one rationale record, which represents a complete explanation (see Figure 2). In this particular case, the record states that since the activity *unix-make* is a known specialization of the activity *update-software-system*, the unexpected action may be a substitution for the more abstract activity node.

An amendment record is next constructed for the explanation which specifies the changes that must be made to the current plan network and domain knowledge in order to restore consistency to the system. The implementation of this rationale record involves replacing the wedge of the plan network subsumed by the more abstract parent node *(update-software-system-01)* with the unexpected action *(unix-make-01)*. As a side effect, the nodes in the expansion of *update-software-system-01* are deactivated from the planner's predictions (see Figure 3).

[8]These implications are derived from relevant plausible inference rules, as described in section 6.3.1.5.3.2.

Unit-name: ALTERNATIVE.ACTION.RATIONALE.01
Unit-comment: "The unexpected action is an alternative
to an in.progress.parent.node of an expected action."
Rationale-summary: "The unexpected action *unix-make-01* is sufficient
since its activity type is a specialization of an in-progress activity node
update-software-system-01. "
Status: complete
Rationale.type: Alternative.action.rationale
Subrationale.type: Specialization.of.in.progress.node
In.progress.parent.activity: update-software-system-01
Pending.goal.achieved: updated(SPANDEX)
Unexpected.action.goal: consistent(SPANDEX)
Hierarchy.level.difference: 2
Amendments: Replace.plan.wedge.01

Figure 2: Rationale record for ACTION.TYPE.MISMATCH.01

Unit-name: REPLACE.PLAN.WEDGE.01
Unit-comment: "Replace a wedge of the plan subsumed by a single node by a new node."
Amendment-summary: "Replace the plan wedge subsumed by
update-software-system-01 with *unix-make-01*."
Implementation: (do
(replace-wedge update-software-system-01 unix-make-01)
(deactivate compile-file-01 compile-file-02 compile-file-03
link-system-01))

Figure 3: Amendment record for ALTERNATIVE.ACTION.RATIONALE.01

6.3.2 *Knowledge Acquisition*

6.3.2.1 Knowledge Acquisition as Knowledge Assimilation

The assimilation of information obtained from domain experts into an existing knowledge base is an important facet of the knowledge acquisition process. Knowledge assimilation requires an understanding of how the new information corresponds to that already contained in the knowledge base and how this existing information must be modified so as to reflect the expert's view of the domain. Here we describe a system, K^n_{Ac}, that modifies an existing knowledge base through a discourse with a domain expert. Using heuristic knowledge about the knowledge acquisition process, K^n_{Ac} anticipates modifications to existing entity descriptions. These anticipated modifications, or *expectations*, provide a context in which to assimilate new domain information.

6.3.2.1.1 Introduction

An often overlooked aspect of the knowledge acquisition process is the assimilation of information presented by the domain expert into an existing knowledge base. Typically, knowledge bases are currently constructed through a series of dialogs between an expert, or experts, in the application domain and a knowledge engineer familiar with the target expert system. The knowledge engineer's task is the modification of the expert system's knowledge base so as to reflect the domain expert's knowledge. To a large extent, this knowledge acquisition task may be viewed as a recognition problem. All of the problems facing other recognition systems are present here as well, including: noisy data (i.e., incomplete or inaccurate information), ambiguous interpretations, and the need to produce intermediate results before all the data is available. Thus, a significant portion of this interactive knowledge acquisition task is a matching problem: How does the expert's description of the domain correlate with the description contained in the knowledge base? How should the knowledge base be modified based on new information from the expert? What should be done when the expert's description differs from the existing one?

K^n_{Ac} is a system that we have developed that implements this knowledge assimilation approach to knowledge acquisition. It was developed to assist in the construction of knowledge bases for the POISE ([27]) intelligent interface system. These knowledge bases use a frame-like representation, described more fully in [53] and [52], to describe *tasks*, *objects* and *relationships* in the application domain. POISE's initial knowledge bases, for the office automation and software engineering domains, were created by hand from interviews between a knowledge engineer and the appropriate domain experts. Transcriptions of these interviews were examined and the results served as the basis of the K^n_{Ac} system.

It is important to note that the goal of the domain expert was *not* to modify POISE's knowledge base; this was the knowledge engineer's role. The expert simply presented the domain information, (e.g., descriptions of tasks, objects, etc.), and responded to questions and comments from the knowledge engineer. The burden of assimilating the information, that is, recognizing where it fit into the existing knowledge base and what additions or modifications were needed, was not placed upon the domain expert. (Contrast this to approaches such as [31], [38], and [48].)

K^n_{Ac} supports the domain expert by trying to assume much of the responsibility for assimilating the expert's information. To accomplish this, K^n_{Ac} models the knowledge engineer's role

EVENT TAKE-A-TRIP-AND-GET-PAID
STEPS: (TAKE-A-TRIP GET-REIMBURSED)
TEMPORAL-RELATIONSHIPS:
((TAKE-A-TRIP *before* GET-REIMBURSED))
CONSTRAINTS: (···)
ATTRIBUTES: ((TRAVELER ···) (COST ···) (DESTINATION ···))

Figure 1: Knowledge Base Event Description

by anticipating modifications to the existing knowledge base using heuristic information about the knowledge acquisition process. As will be described later, these anticipated modifications allow K^n_{AC} to focus on "relevant" portions of the knowledge base and provide a context in which to integrate the information provided by the domain expert.

Consider the opening portion of a discourse in which the expert, the principal clerk of an academic department, is describing the procedure for being reimbursed for business-related travel expenses.

> "O.K. — on travel. The proper way of doing it, if it's out of state, is that a travel authorization should be issued before the trip."

From this information one can conclude that some unnamed task consists of two temporally ordered steps. Rather than simply adding this information to the knowledge base, it may be desirable to modify an existing task description. However, it is not clear what modifications need be made to the knowledge base to reflect this additional information.

If the knowledge base (prior to this interview) is examined, a description of the reimbursement process will be found (see Figure 1). In this simplified view of the task, which knows nothing about a "travel authorization," the traveler simply goes on a trip and gets reimbursed. Though the knowledge engineer may realize that the clerk and this existing description are describing the same task, it is not readily apparent from the two descriptions. Matching such descriptions, and recognizing the implied modifications, are central to the assimilation process.

To accomplish the assimilation of this new information, K^n_{AC} was required to perform two basic tasks: 1) recognizing where the expert's information fits into the existing knowledge base, and 2) appropriately modifying the existing knowledge so that it reflects the expert's view of the domain. Determining where the expert's information fits into the existing knowledge requires that the new information be matched against the existing information. To avoid matching the new information against the entire (existing) knowledge base, the most likely candidate matches must be selected. Furthermore, since the goal of a knowledge acquisition discourse is the modification of the knowledge base, exact matches between the new and the existing information are not always expected.

Thus, the procedure for matching the expert's entity descriptions with those already in the knowledge base must be specialized for knowledge acquisition. K^n_{AC}'s matching and match evaluation procedures are described in Section 6.3.2.1.3. Discrepancies between the expert's descriptions and the existing ones may imply needed modifications and need not degrade a match, especially if the discrepancies (or the implied modifications) can be predicted. Anticipated modifications, or *expectations*, arise from an understanding of the knowledge acquisition process.

They can be derived from the state of the existing knowledge base, from cues in the discourse, from previous modifications to entity descriptions, or from the state of the knowledge acquisition task. The generation and management of these expectations is described in Section 6.3.2.1.4. Finally, the status of this work and its contributions to the knowledge acquisition task are summarized in Section 6.3.2.1.5.

6.3.2.1.2 The K^n_{AC} System

In this section, the basic architecture and functionality of the K^n_{AC} system[9] is presented. During each cycle of the K^n_{AC} system, descriptions of domain entities are accepted from the user (1) and compared with entities in the existing knowledge base (2). (Figure 4 contains a portion of this knowledge base.) These candidate entities (3) are selected based on K^n_{AC}'s expectations of changes to the knowledge base. The comparisons (4) are evaluated both in terms of how well they match and the extent to which the differences between them were expected (5) within the context of the match. Once the best matches are selected, the implied modifications (6) are made to the existing entity knowledge base (7), after being verified with the user (8), if necessary. Expectations of further modifications are generated from a variety of sources, including the information obtained from the discourse (9), the state of entities in the knowledge base (10), previously made modifications (11) and the state of the acquisition process.

The descriptions obtained from the expert must be presented to the matcher in the knowledge base's representation language. The purpose of the *discourse manager / user interface* is twofold: to permit a more "user-friendly" specification (e.g., natural language, graphics, menus, etc.) of these descriptions, and to provide K^n_{AC} with any available cues as to the state of the discourse. Note that the focus of this work is not the translation of the expert's knowledge into the representation used by the underlying knowledge base. Rather, K^n_{AC} addresses the issue of how to integrate this knowledge once it is in such an accessible form. Currently, the natural language protocols are translated by hand into the system's representation language. Only minimal discourse cues, such as "topic" information, are assumed to be available.

Thus, the discourse fragment presented in Section 6.3.2.1.1 translates, approximately, into the structures shown in Figure 3. These structures may then be compared with selected entities from the existing knowledge base.

To avoid having to examine the entire knowledge base in order to assimilate the new information, entities that are most likely to be modified are selected as candidate matches. Thus, if there exists an expectation of some modification to a given entity description, that entity is compared to the new information from the expert. The way in which these modifications are anticipated will become clearer in Section 6.3.2.1.4. Initially, the only expectations available are based on the discourse cue recognizing that TRAVEL is a topic of interest. Hence, the entities semantically close to TRAVEL in the knowledge base are selected as candidate matches. These entities include TAKE-A-TRIP-AND-GET-PAID, shown in Figure 1.

The system then compares the expert's descriptions with the selected match candidates. The matching process, described more fully in Section 6.3.2.1.3, determines the similarities and differences between a pair of entity descriptions. The results of these comparisons are then evaluated in order to select the best match for each of the expert's entity descriptions. Section 6.3.2.1.3.2

[9]Figure 2 contains the architecture of the K^n_{AC} system. The parenthesized numbers in this paragraph (e.g., (1)) refer to this figure.

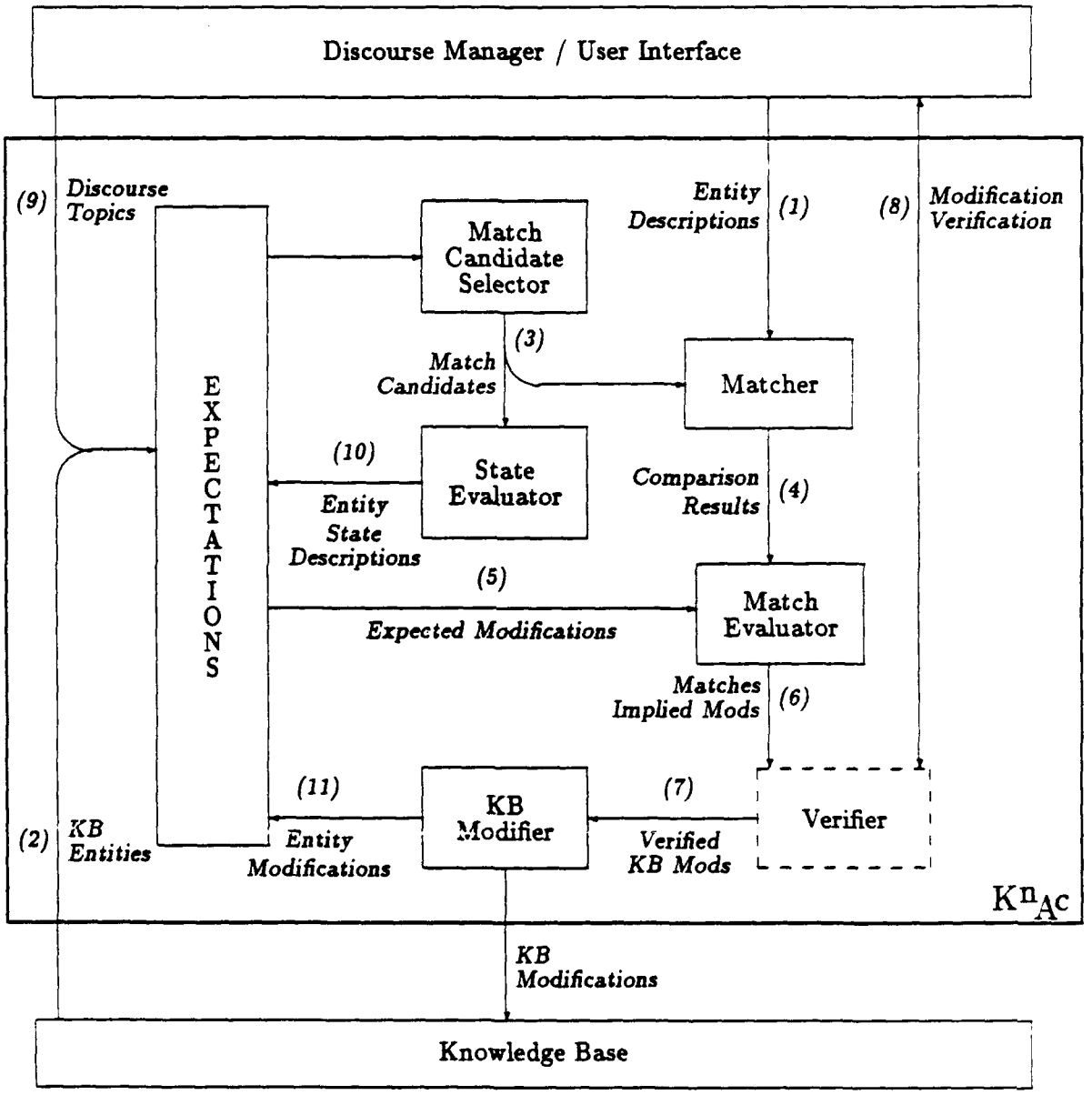

Figure 2: The K^n_{Ac} System Architecture

EVENT EVENT-1
STEPS: (ISSUE-TRAVEL-AUTHORIZATION TAKE-A-TRIP)
TEMPORAL-RELATIONSHIPS:
((ISSUE-TRAVEL-AUTHORIZATION *before* TAKE-A-TRIP))
CONSTRAINTS: ((DESTINATION *outside-of* STATE))
ATTRIBUTES: ((TRAVELER ···) (DESTINATION ···))

EVENT ISSUE-TRAVEL-AUTHORIZATION

EVENT TAKE-A-TRIP

OBJECT TRAVEL-AUTHORIZATION

Figure 3: Discourse Manager Output

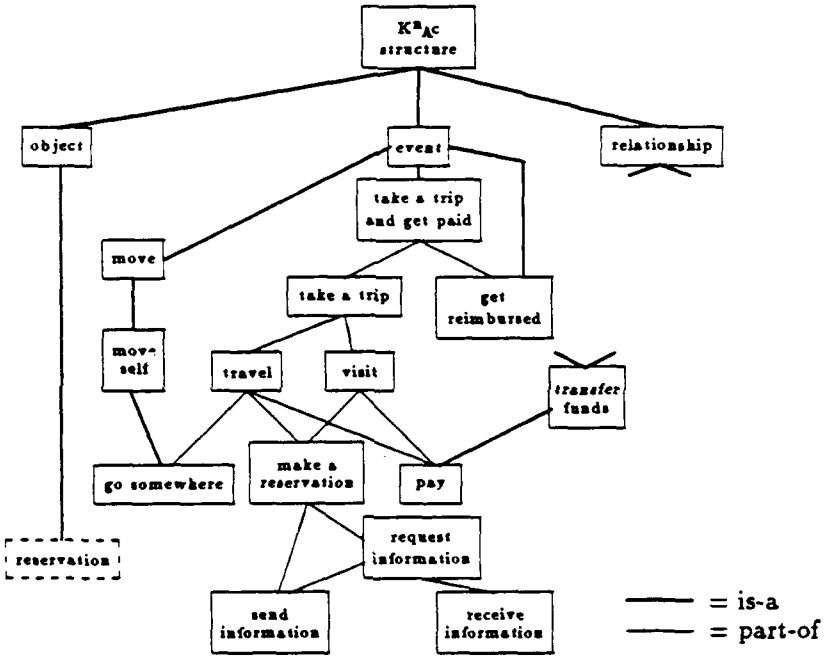

Figure 4: A portion of the knowledge base

describes the evaluation process, which rates the match results on a field-by-field basis and combines these ratings to produce an overall rating for each match.

For example, when the unnamed task described by the expert, labeled EVENT-1, is compared with the existing task description TAKE-A-TRIP-AND-GET-PAID, the contents of each field of these structures (e.g., *parts*, *generalizations*, *temporal-relationships*, *pre-* and *post-conditions* etc.) are compared. In the *steps* field, they have one entity in common (TAKE-A-TRIP) and each has one entity not found in the other (ISSUE-TRAVEL-AUTHORIZATION in EVENT-1 and GET-REIMBURSED in TAKE-A-TRIP-AND-GET-PAID). They have several *attributes* in common (TRAVELER and DESTINATION). Their *temporal-relationships* are mutually consistent, though different. Both entities are *specializations* of EVENT.

The ratings for these field matches is shown in Figure 5. Remember, these ratings reflect not only the degree to which the fields match, but more importantly (from the point of view of

EVENT-1 vs. TAKE-A-TRIP-AND-GET-PAID

Field	Rating
generalizations	1.000
parts	0.133
attribute-names	0.055
constraints	0.007
Match Rating	0.299

Figure 5: Ratings for field matches

knowledge acquisition) the likelihood of the modifications, in future parts of the dialog, required to make them match.

From these ratings, the best matches are selected. If there is significant ambiguity, the matches are verified with the expert. If there are no acceptably good matches for one of the expert's entity descriptions, a new entity is added to the knowledge base.

When the best matches have been selected, the differences between the expert's description and the existing one are used to modify the knowledge base. For instance, the "extra" step in EVENT-1, (i.e., ISSUE-TRAVEL-AUTHORIZATION, is added as a step of TAKE-A-TRIP-AND-GET-PAID). The extent to which this modification requires confirmation from the expert depends on the level of autonomy granted to the system. At various levels, all such modifications could be verified with the expert or only unexpected ones or only deletions, etc.

Once the knowledge base has been modified, new expectations are generated to be used in interpreting the next "discourse frame." The generation and management of these expectations is described in Section 6.3.2.1.4.

6.3.2.1.3 Matching for Knowledge Acquisition

In order to match entity descriptions provided by the domain expert to those already known to the system, K^n_{Ac} must be able to compare these structures and evaluate the results. This matching process, while in some ways similar to that found in most recognition/interpretation systems, displays certain characteristics unique to knowledge acquisition. In particular, since the goal of a knowledge acquisition dialog is the modification of the knowledge base, the information provided by the domain expert should not completely match the existing entity descriptions. The matching process must be modified so as to be able to recognize and, where possible, anticipate these discrepancies. The matching techniques presented in this section make these discrepancies explicit; the evaluation of these match results, described in Section 6.3.2.1.3.2, incorporates the extent to which these discrepancies were anticipated into its rating procedure.

6.3.2.1.3.1 Matching Entity Descriptions

POISE's knowledge base is represented in a frame-based language, similar to that used by systems like Knowledge Craft® ([90], [15]). Comparison of these knowledge structures requires matching on a field-by-field basis. Each field may be considered to be one of two types of structures: a *set of elements* such as *steps* and *generalizations* of an event, or a *collection of constraints* such as the *temporal relationships* and the *pre-* and *post-conditions*. K^n_{Ac} contains matching techniques

for each of these types of structures.

Set Matching

Determining how well two sets of elements match is not difficult; neither is determining how "different" they are (e.g., see [82]). For knowledge acquisition purposes, however, the relevant question is *"How likely is it that they can be modified so as to match?"* To determine this, K^n_{AC} examines not only the elements in each (set) field of the knowledge structure, but also makes use of information *about* that field (i.e., *meta-information* or *facets*). In particular, information about the size of each field and the range of the elements in that field permit K^n_{AC} to calculate the probability that the *extra* elements in one of the structures will be added to the other.

Consider the comparison of the *steps* of EVENT-1 and TAKE-A-TRIP-AND-GET-PAID. A typical measure of similarity is:

$$\text{match-rating} = \begin{cases} 1 & \text{if } \text{Set}_1 = \text{Set}_2 = \emptyset \\ \dfrac{|\text{Set}_1 \cap \text{Set}_2|}{|\text{Set}_1 \cup \text{Set}_2|} & \text{otherwise.} \end{cases}$$

With one step out of the three unique steps between them in common, the similarity of these fields would be 1/3. How likely is it, however, that the "extra" step in the expert's description (i.e., ISSUE-TRAVEL-AUTHORIZATION) will be added to the existing description? Without requiring a deep understanding of domain-specific semantics, some additional information can still be used. If events usually have few steps (as specified by the meta-information about the *step* field of EVENT), the addition of this particular step is less likely than if there are many more steps still to be added. Similarly, if a step is going to be added, the range of possible steps will affect the probability of the desired one being added. This range is determined by combining the "type restriction" meta-information about the slot (i.e., a *step* of an EVENT must be an EVENT) with the existing knowledge base (i.e., the number of EVENT descriptions known to the system).

Thus, K^n_{AC} rates each set match based on the likelihood of the modifications implied by the match. The likelihood is determined from the anticipated size and range of the sets and requires no additional semantic information about their content. The derivation of these ratings is fully described in [52].

Constraint Matching

Although comparing (or combining) arbitrarily constrained sets of entities is a difficult problem often requiring substantial domain knowledge, K^n_{AC} compares sets of constraints, pairwise related by a common relation, in a purely mechanical fashion.[10] This section describes a mechanism for comparing such constrained sets; the mechanism is independent of the particular relation, requiring only a description of its algebraic properties, (i.e., whether or not the relation is *reflexive*, *symmetric* and/or *transitive*).

First, any implicit constraints are made explicit by propagating the specified constraints using the relation's algebraic properties. The temporal relation *before*, for example, is only transitive; if the constraints (A *before* B) and (B *before* C) were specified, then (A *before* C)

[10]The current system checks each relation separately; it does not handle interaction between different relations, though it is able to combine relations with their inverses. For instance, *before* and *after* constraints are handled together.

could be deduced. When the constraints are thus propagated, inconsistencies may be detected by checking the results for any of the prohibited properties of the relation (i.e., non-reflexive, non-symmetric and non-transitive). If each set of constraints is internally consistent, the two sets may be merged and re-propagated, and this combined set of constraints may be checked for consistency.

If two sets of constraints are mutually consistent, a measure of their similarity may be obtained by determining the changes required to make them equivalent. Simply adding each set of constraints to the other would accomplish this, but this may add more information than is necessary. For instance, if the first set of constraints contained (A *before* B) and (A *before* C) and the second sets contained (B *before* C), then only (A *before* B) need be added to the second set. Requiring the addition of both constraints from the first set would imply a larger discrepancy between the sets than actually exists. Obtaining the minimal set of constraints that need to be added is not a trivial problem. $K^{n}_{A}c$ contains a new approach to generating these sets, called "opensures"[11], based on their algebraic properties.

Thus, as with fields containing *sets*, $K^{n}_{A}c$ is able to rate *constraint* matches by determining the likelihood of the implied modifications. This rating is based on the algebraic properties of the relationships involved and requires no additional semantic information. $K^{n}_{A}c$'s methods for constraint propagation, determination of consistency, and generation of opensures are presented in [52].

6.3.2.1.3.2 Match Evaluation

After comparing each of the entity descriptions provided by the domain expert against those candidate entities selected from the existing knowledge base, $K^{n}_{A}c$ evaluates these match results in two passes. First, the likelihood of two entities matching, based on the extent to which they differ and the probability of these differences being corrected, is determined as described above. This "degree of fit" is a relatively inexpensive means of pruning the set of possible matches.

The second pass of the match evaluation takes into account the context in which the comparison is being made. In addition to *how* the descriptions differ, it considers whether these differences are expected in a particular situation. The extent to which the modifications implied by the differences between the structures are expected serves as a "context-dependent" measure of the match. The anticipation of such modifications is a crucial part of the $K^{n}_{A}c$ system and is described in the following section.

Consider, for example, the addition of the step ISSUE-TRAVEL-AUTHORIZATION to the description of TAKE-A-TRIP-AND-GET-PAID. The addition of such a step could have been foreseen for several reasons. First, since there were fewer steps in the existing description than are typically found in events, adding another step was reasonable. More importantly, upon examining the existing description to see if it was consistent and complete, it was discovered that a *precondition* of the step GET-REIMBURSED, describing the traveler as the recipient of funds, could not be satisfied by the only earlier step in the task (i.e., TAKE-A-TRIP). This further supported the addition of another step (occurring before GET-REIMBURSED) to the task.

[11] i.e., the inverse of "closures"

6.3.2.1.4 Anticipating Modifications

K^n_{Ac} provides a context in which to interpret information provided by a domain expert by anticipating modifications to an existing knowledge base. These anticipated modifications, or *expectations*, are derived from K^n_{Ac}'s heuristic information about the knowledge acquisition process. This section describes how these expectations are generated, how they are used to provide a context in which matches may be evaluated, and how they ranked and managed.

6.3.2.1.4.1 Generating Expectations

K^n_{Ac} contains a body of heuristics about the knowledge acquisition process. These heuristics, obtained through the analysis of several knowledge acquisition dialogs, allow K^n_{Ac} to anticipate modifications to an existing knowledge base. These heuristics may be divided into four categories. The first group is based on the state of the knowledge, both that already contained in the knowledge base and new information provided by the expert. The second category depends on modifications previously made to the existing knowledge base. The third set makes use of a model of the discourse process, while the final set incorporates knowledge about teaching and learning strategies.

Since it is expected that the collection of heuristics will be modified, both as a result of improved understanding of the knowledge acquisition process and through the addition of domain specific heuristics, K^n_{Ac} allows heuristics to be added (or removed) in a straight-forward way. Most of K^n_{Ac}'s current heuristics are quite simple and domain-independent; the addition of more complex, application-specific heuristics may improve the system's performance in certain domains.

K^n_{Ac}'s *state* heuristics are based on the assumption that the information contained in the knowledge base should (eventually) be *complete* and *consistent*. Obviously, such attributes can be measured in a variety of ways, some dependent on the application domain, some dependent on the knowledge representation, and some rather generic. Consider a simple heuristic based on one measure of incompleteness:

Heuristic S2: *Fields with too few components will be augmented.*

This heuristic states that if information is detected to be missing from a field of some entity description, the addition of that information may be expected. One simple approach to detecting such missing information compares the number of entries in a field of a knowledge structure with the field's expected cardinality. If the field is determined to contain too few values, additional values (of the appropriate type) will be expected. The expected field size may come, in order of specificity, from meta-information about a given field of a given entity, via inheritance from a generalization of the entity, from the default information for the type of the entity, or from an overall field default size. This size information may be static or determined dynamically by the system. An expectation generated by this heuristic is:

```
Exp146:  Expecting (certainty 0.360):
   MOD:  ADD ?New-part<is-a-knac-structure-p> to the
         Parts field of Take_a_trip_and_get_paid
         Derived from Take_a_trip_and_get_paid and H_S2.
```

Other *state* heuristics exploit references to unknown entities, unsatisfied preconditions of task steps, and range/value conflicts to detect inconsistencies in the knowledge base and anticipate changes. One example of such heuristics,

Heuristic S3: *Unsatisfied step preconditions will be satisfied.*

uses incompleteness in an event description to anticipate the addition of a step and a temporal constraint placing it before the existing step with the unsatisfied precondition.

In addition to the state of the knowledge, changes made during the knowledge acquisition process may imply additional modifications. For instance, two *modification* heuristics are triggered when a new entity description is added:

Heuristic M1: *Detailed information usually follows the introduction of a new entity.*

Heuristic M2: *Context information usually follows the introduction of a new entity.*

Additions of information to specific fields of entity descriptions (e.g., attributes, steps, constraints, etc.) form the basis for other more specific modification heuristics. Examples of such heuristics include:

Heuristic M5: *Adding two parts to an entity usually implies a relation between them.*

Heuristic M4a: *Parts of Events are usually temporally constrained after being introduced.*

Cues from the discourse manager, such as the topic of discourse or recently referred to entities, are the key to K^n_Ac's *discourse* heuristics. Because the current system lacks a sophisticated discourse manager, we do not rely heavily on discourse cues. Thus, the only discourse heuristics being used are:

Heuristic D1: *Entities close to specified topics are likely to be referenced or modified.*

Heuristic D2: *Referenced entities are likely to be modified or referenced again.*

These heuristics generate expectations of some unspecified modification to the referenced entities or to those semantically close to them. "Closeness" is determined by the number and types of relationships (i.e., links) separating two entities.

6.3.2.1.4.2 Managing Expectations

As the number of expected modifications to the knowledge base grows, K^n_Ac's ability to use the expectations to focus its attention diminishes. Thus, a means of selecting the most likely expectations (for a given point in the acquisition discourse) is required. This is accomplished by assigning a rating to each expectation and pruning the set of heuristics based on this rating.

Each heuristic is responsible for determining a rating for each expectation it generates. This rating depends on the quality of the data and the specificity of the heuristic. Since different types of heuristics generate expectations based on different types of data (e.g., state information, previous modifications, etc.), each heuristic has its own function for determining ratings. For example, Heuristic D1 (shown above) includes the semantic "distance" from the specified topic in its rating calculation; Heuristic S2, which is based on "missing information," incorporates the system's certainty that the information is actually missing.

In addition to the initial ratings assigned to each expectation, each heuristic contains a function that specifies how these ratings will change with time. For instance, certain expectations are important when they are created but become less valid with the passage of time; others become more critical as time passes. Some become more (or less) significant based on some state of (or change to) the knowledge base; others are always valid. $K^n_A c$'s current set of predefined functions for specifying the change in rating includes: *fade*, *increase*, *until*, *while*, *after*, *for*, *always* and *never*.

6.3.2.1.5 Status and Conclusions

In this chapter we have examined an often overlooked aspect of the knowledge acquisition process: the assimilation of information presented by a domain expert into an existing knowledge base. Though a fundamental part of the current conventional knowledge base development process, the issue of automatically locating and appropriately modifying existing knowledge to conform to the domain expert's descriptions has received little, if any, emphasis. Most current knowledge acquisition tools place this burden on the domain expert, forcing him to take over part of the knowledge engineer's task. By automating this assimilation process, the $K^n_A c$ system better insulates its user from the knowledge base.

$K^n_A c$ accomplishes this assimilation by: 1) comparing entity descriptions provided by the domain expert with existing knowledge base descriptions, 2) evaluating these matches in the context of the knowledge acquisition discourse, 3) making the modifications to the existing descriptions implied by the expert's information, and 4) generating (and managing) expectations of further changes to the knowledge base.

This work has developed several generic matching (and match evaluation) techniques especially adapted for knowledge acquisition. They shift the focus of matching from examining how closely two entities match to exploring the likelihood of their being modified so as to match. This is accomplished by matching procedures (for sets of entities and collections of constraints) which determine differences as well as similarities, an evaluation technique which explores the probability of the modifications required to make entities match and the degree to which these modifications are expected, and a means of anticipating modifications to the knowledge based on heuristic information about the knowledge acquisition process.

The $K^n_A c$ system is implemented in Common Lisp running on a TI Explorer®. Its current use is still experimental, though it has been able to assimilate a 20 step dialog on the travel reimbursement process, correctly constructing an internal representation of the plan in the POISE knowledge base. A complete description of the implementation and the sample dialog can be found in [52].

6.3.2.2 Knowledge Acquisition For Planners

Previous work in knowledge acquisition has primarily addressed rule based expert systems and concept hierarchies in knowledge bases. The special constraints in the acquisition of knowledge for planners have not been explicitly identified. Here, we identify the requirements that situation calculus-based planners pose on the knowledge acquisition process and compare these requirements to the techniques currently used or proposed for rule-based expert systems. We then propose a framework for the human representation of subject performed tasks. To verify the validity of the framework, we conducted several experiments with more than 150 subjects. The last part of this chapter describes the DACRON plan acquisition interface which is based on the framework.

6.3.2.2.1 Introduction

The majority of research in knowledge acquisition has concentrated on the acquisition of rules and concepts for rule-based expert systems [58, 48, 8]. Domain-independent planners [54] are another type of knowledge-based system whose performance in real environments depends critically on knowledge acquired from end users. Despite this, little attention has been paid to plan acquisition. The knowledge in a planning system is different both in its structure and its use than the IF-THEN rules in expert systems. The basic unit of plan knowledge, sometimes called a plan schema, contains information about the goal, preconditions, subgoals and effects of the action it represents. Figure 1 shows a plan schema from the POLYMER planning system developed at the University of Massachusetts [22]. It describes how to purchase an item and specifies causal dependencies, responsible agents, constraints, and affected objects as well as the goal and precondition information.

```
goal:            have(item,self)
decomposition:   fill-out(order-form), file(copy), mail(order-form),
                 receive(item), pay(invoice)
precondition:    have(money,self), have(order-form,self)
side-effects:    have-less(money.self), have(copy.self)
causal depend.:  enables(fill-out(order-form), file(copy)),
                 enables(fill-out(order-form), mail(order)),
                 enables(mail(order), receive(item)),
                 enables(receive(item), pay(invoice))
agents:          purchase-clerk
objects:         order-form, item, copy, money
constraints:     item-price < money-available
```

Figure 1: Purchase task as POLYMER plan.

To acquire plans, we have to carefully review the stages of knowledge acquisition, including explaining to a domain expert what we want to acquire, eliciting the knowledge, coding the knowledge, displaying the knowledge, and debugging the knowledge. The psychology of pro-

gramming [64] gives us a framework for defining the stages of plan knowledge acquisition and discussing the differences between acquisition for planners and for expert systems. This is done in the next section.

In order to build a plan acquisition system, we have to have a model that links the elicitable knowledge of the domain experts to the requirements of the planner. Having such a model allows us to design the plan acquisition system with some confidence that domain experts will be able to use it. We present a model for the recall of plans in Section 6.3.2.2.3 and discuss the implications of the model in Section 6.3.2.2.4. In Section 6.3.2.2.5, we present the DACRON plan acquisition system which is based on the approach outlined. DACRON is designed to be used in conjunction with the POLYMER planning system, and issues such as the construction of plans, execution of plans, and handling of exceptions are POLYMER's responsibility. The main goal of DACRON is to allow domain experts to specify plan knowledge that can be used as a starting point, even if it is incomplete or incorrect. The final section discusses our future research plans.

6.3.2.2.2 Knowledge Acquisition: Planners versus Expert Systems

We view knowledge acquisition as an instantiation of general programming techniques. The psychology of programming can therefore be modified to accommodate knowledge acquisition [64]. Pea and Kurland identify four stages in the process of software development:

- understanding the problem;
- solving the problem;
- coding the solution;
- debugging the code.

The problem that has to be solved and coded in the general software environment by a programmer corresponds to a particular piece of domain knowledge that has to be acquired, (i.e. specified by a domain expert to the acquisition system). The programmer understanding the problem therefore corresponds to making the domain expert understand what particular knowledge the system wishes to acquire.

Solving the problem corresponds to the domain expert recalling a piece of knowledge. Gruber coined the term *acquirable form* for knowledge in this stage [40].

The actual process of coding this knowledge (operationalization) means translating the acquirable form into a form meaningful to the system. In general programming this means transforming an algorithm to a programming language; in the case of knowledge acquisition it means expressing the acquirable form in terms of primitives of the acquisition system.

Once the knowledge is operationalized, it might be displayed in a different way than it was entered. This adds another stage to the process of knowledge acquisition. Users have to understand what they actually have specified. Only then can they understand how this new piece of information is viewed by the system, which in turn enables them to debug it.

We can summarize the stages of knowledge specification, which will serve to identify differences in the knowledge acquisition process between expert systems and planners:

- Request: Description of particular knowledge to be acquired;

- Elicitation: Recalling knowledge;
- Operationalization: Coding the recalled knowledge;
- Display: Echoing the coded knowledge;
- Debugging: Correcting the coded knowledge.

To acquire a rule, most existing knowledge acquisition systems for expert systems start at the first stage of this theory and follow the stages sequentially for each rule that is to be acquired. To acquire a single plan, domain experts have to move back and forth between these stages. Users might start the operationalization of the goal, move on to the request stage of the decomposition and then debug the precondition of a single plan.

Rules usually express a causal or temporal relationship. Plans usually consist of more elaborate information. Therefore units of knowledge are smaller in expert systems than in planning. This fact facilitates the request phase in expert systems.

Rule based expert systems have to elicit the IF part of the rule and the THEN part of the rule. Once the experts have recalled the knowledge that corresponds to each of these parts of the rule, they can start to specify it. Techniques used to accomplish this elicitation are usually based on association, comparison or cueing [8, 58]. As plans have a more complex structure than rules (goal, subgoals, precondition, constraints, temporal and causal orderings, etc.), the elicitation process must be less uniform. Additional requirements for plan acquisition are a clear conceptual distinction between components of the plan, specific techniques to elicit particular components and an overall framework to integrate the components. Techniques developed for rule based expert systems, like repertory grids, heuristic classification, certainty ranking or choice annotation are of no help in this case. Certain forms of cued recall and structured frames seem to be most appropriate for the various demands in plan acquisition [56].

The largest problem in both areas is the operationalization of the elicited knowledge. The techniques applied most commonly are editors that specialize in the syntax of knowledge representation formalisms. This technique will not work as well in the acquisition of plan knowledge from domain experts because of the conceptually different slots of plans and the experts lack of knowledge representation skills. In planners, each slot needs to be addressed in a semantically correct way that is also immediately understandable to the domain expert. We found that electronic forms questionnaires as mentioned by Gruber and by Musen in conjunction with iconic representation of objects work best to fulfill all these requirements [40, 62].

Specified knowledge can either be displayed in the same form that it was entered or it can be set into the context of other knowledge in the system. For expert systems, this usually means displaying a rule network. The different reasoning mechanism in planners leads to a different display strategy, namely that of a hierarchy. Such a component is mandatory for successful debugging.

The above discussion shows that the acquisition of plan knowledge is considerably different than the acquisition of rule knowledge. In addition, there exists no model linking the humans subjective view of a task to a knowledge representation formalism. In this work, we try to build a plan acquisition system based on the implications of these differences.

6.3.2.2.3 A Framework for the Recall of Tasks

We would like to acquire plan knowledge from domain experts about the way they execute tasks. A task is an activity which is viewed by the human as a unit at a certain level of abstraction (e.g., *purchase an item* or *receive reimbursement for travel*). This focus eliminates many existing models of human cognition as possible candidates because either they make no predicitions on the task level [2, 3, 72], or they evaluate work done with computer systems themselves [14, 66] or they address interaction and discourse issues only [77, 78, 74]. None of these theories makes statements about structures and processes involved in the recall of complex tasks from long term memory. Therefore, we started a series of interviews and experiments to build a framework for complex task recall. Interviewing domain experts about how they conduct certain complex tasks usually leads to a description of an example. In these examples, we observed the grouping of operations, which are the lowest level of description, into units. These units fit what is called *Handlung* by German psychologists and philosophers. A *Handlung* is a *conscious, goal-directed act of a human being, controlled by will, directed towards shaping reality. It contains three aspects: an intended goal, an analysis of means for its achievement and the decision to do so.*[12] A *Handlung* contains operations, conducted by a human, which transform states of reality into other states, serving a certain purpose [50]. (Clearly a *Handlung* is more complex than the English equivalent *act*. In the remainder of this chapter we will refer to it as act, though).

The act is the smallest coherent unit in the description of a task that appears to be at the appropriate level of abstraction. This individually perceived appropriateness of the act as the smallest unit, varying from person to person and from task to task, makes the concept very suitable for our purpose. It also distinguishes the act from GPS-operators [63]. While operators are established by task analysis, acts are the representation of the human's perception of these tasks.

As the above definition of acts does not lend itself to immediate operationalization, we try to formalize its properties in information processing terms [73]. The representation of an act consists of four properties which are discussed below.

The first property of an act is the conscious goal; in our case it is the intention to complete a certain task. In our staged theory of knowledge acquisition, this intention is equivalent to understanding what is to be acquired. During work on one task, this goal remains the same, only situations (states) change, not intention. People consciously know about this goal and should be ready to report it without difficulty.

The current state of reality, as mentioned in the above definition [50], constitutes the second property. We call this property the pre-situation. During execution of the same task (same goal) the situation determines which operations are to be applied. Only when the task is completed or interrupted does the goal change. When people are actually performing a task, they directly perceive the pre-situation. During recall, the content of the working memory mirrors this state. People who are imagining working on a certain task should be readily able to report what the current state of affairs is.

The third property of the definition is the decision to generate behavior, which in turn is observable. We represent this as a list of names of the operations to be generated. As we are not concerned with actual execution of tasks, but only the part of them that can be reported by recall, we must be satisfied with the name the person ascribes to a certain operation. The

[12] Definition from: Der grosse Brockhaus; 17th Edition. Translation by the authors.

mapping of theses names to primitives of the system is a question of coding and will thus be discussed in the next section.

The result is also available in the recalled act. We call this fourth property, describing the situation after the application of operations, the post-situation. The post situation is described in the same terms as the pre-situation but it includes the changes caused by the operations. The justification for this property comes from the explanation that acts transform states of reality into other states of reality.

The complete representation of a formal act is given by the structure in figure 2. We are aware that the operationalization of such a complex concept as an act can not capture every nuance in meaning and must fall short in certain aspects of description. The structure we present here will certainly have to be amended. Acts are the product of recall of deeper cognitive structures.

Goal: Intention of the Whole Act
Pre-Situation: List of Properties
Operations: List of Operations
Post-Situation: List of Properties

Figure 2: Operationalization for acts

We assume that recall can be directed to decompose and sequentialize acts. Sequentialization is the process of finding a sequence of acts that leads from one state of the world to another one. Decomposition is the process of breaking an act into lower level acts, making the higher level act the goal and generating all its constituting operations as lower level acts. As our framework is concerned with the existence of recall processes and structures and not with low level, underlying cognitive processes, we make no assertions as to how this is accomplished. It is sufficient for our purposes to know that these processes and structures exist at all, not how and why.

In addition to the recall of activities, we assume the recallability of objects, relations and states of the world. As these entities comply to a certain degree with those items usually encountered in recall experiments, we need make no additional assumptions.

With the operationalization of acts and the proposed subprocesses, we were able to propose and test the following hypotheses about the recall of subject performed activities from long term memory. It is important to realize that these are only one set of many possible descriptions for the phenomena occurring but that they seem to be a reasonable one for our purpose.

1. A general task (goal) and a specific situation (pre-situation) will result in the recall of a specific set of operations.

2. Changing the pre-situation or the general task (goal) will result in the recall of different operations.

3. A goal, a start situation and an end situation will result in the recall of a sequence of acts. This sequence leads from the start situation over intermediate situations to the end situation.

4. An operation may itself be composed of acts and those acts are recallable (*decomposition*).

5. If the pre-situation of an act is not achievable, other acts with the same goal but different pre-situation can be recalled instead (alternative acts).

6. Operations can be distinguished from the post-situation created by these operations.

7. Causal dependencies between operations can be indicated.

To test these hypotheses, we first conducted a pilot study with four departmental secretaries and then a larger series of experiments. Our subjects in those experiments were 153 undergraduate and graduate students at the University of Massachusetts and Smith College. We could verify all hypotheses. Special care had to be taken with hypotheses number six (distinguishing post-situation from operations). Subjects were able to report the post-situation if asked directly about it. They could not report the post-situation if queried in general terms. For a detailed discussion of the experiments and their results see [56].

6.3.2.2.4 Implications of the Framework for the Acquisition of Plan Knowledge

The properties of the act framework can be used to create design criteria for the plan knowledge acquisition system.

The most important implication of the framework is that humans react to a situation differently with different intentions and that they do not set up recursive goal stacks but go step by step. People are guided by context and fine tuned by the current situation. In the explanation phase, a domain expert could be instructed to specify a certain piece of knowledge by just stating the goal of the plan.

The human description of a plan consists of a sequence of acts. This acquirable form can be directly translated from the elicitation phase to code. A plan definition system should allow the specification of each of these acts separately. The decomposition of acts into primitive plans and subgoals follows as a consequence of the primitives of the system. The act is either a primitive plan the system understands, or it has to be explained in terms of those primitive plans.

The problem of breaking a plan down into hierarchies of goals and subgoals should therefore not be forced on the user but evolve from the user's perspective. Decomposition and sequentialization are the processes to employ in this situation. The experiments show that a complex act can be decomposed. People can also give a sequence of acts from a start state to an end state. A combination of both techniques, decomposition and sequencing, seems to be most appropriate for the specification of complex plans.

People deal with one act at a time. The system should therefore not force the user to specify the sequence of acts as a whole. The specification of a new act should only begin when the last one is completely defined. This guarantees completeness and consistency of the individual pieces of knowledge and supports the human desire for closure.

The versatility of a planner can be greatly enhanced if there are numerous plans that achieve the same goal under different preconditions. People are able to report various acts to achieve a goal. The list of operations and the different preconditions are the major difference between these alternative acts. A plan specification system should solicit as much information as possible. It should therefore in the process of acquiring a plan also acquire alternative plans that meet the same goal.

Our experiments show that the specification of effects is more dependent on the interaction technique than others. People seem to assume that the effects of an operation or act are implicitly understood by the system. Though people can answer all questions about effects correctly, they cannot report and specify them easily. An approach where users have to fill in forms or create forms yielded the best results.

The last property of the model was the indication of dependencies between operations. Our experiments show that though people think of and report their operations sequentially, they can indicate causal dependencies and thereby provide information for the parallel execution of plans. The specification of dependencies is best done graphically. Arrows indicating causal dependencies between acts are more easily understood than verbal or other methods.

6.3.2.2.5 The DACRON Plan Specification Interface for Domain Experts

DACRON is a plan acquisition system with an emphasis on immediate usability for the domain expert. DACRON addresses every aspect of the knowledge specification process introduced in section 2. We developed DACRON on the implications of the framework for task recall. Our goal was a system that would help domain experts to specify their knowledge in a way closest to their individual view of operational knowledge (acts) in the domain.

DACRON provides users with an *act type* to specify tasks. The act type appears as an icon. This icon can be opened and presents the user with a graphic form editor (figure 3). The editor shows compartments that refer to the pre-situation ("before-compartment"), the operations ("action-compartment") and the post-situation ("after-compartment") of the particular act, which correspond directly to the same entities in the framework. These compartments are ready to accept other icons as input.

The second construct is the *object/relation type*. Object/relation types represent objects and relations, which are used in the description of tasks. Object/relation types appear also as icons and can be opened to present their code in a form editor. The editor for object/relation types shows rows to hold names for arguments and primitive data types for arguments (figure 3). The icon for the object/relation type differs from that for the act type, as figure 3 shows, in that the act icon has extensions for arguments. Icons carry a label to make them distinguishable.

To specify a new task, users can either copy and modify an existing act icon or begin with a blank one. Users start by naming the act type (e.g., *purchase-item-1* (figure 4)). Then they can specify the goal by moving one or more object/relation types to the goal compartment and setting the arguments of these types to the desired goal-values, for example: moving the icon for *item* to the goal compartment, opening it and setting the value of *owner* to *self* (figure 4). Moving object/relation icons into the "before compartment" (pre-situation) and the "after compartment"(post-situation) and act icons into the "action compartment" (operations) specifies those slots of the plan. Causal dependencies between acts in the action compartment are indicated by arrows. Constraints are placed in the compartment that contains the object to which a certain constraint applies. In the case of inter-object constraints which might apply to objects in different compartments the users might choose the more convenient one.

Another implication of the framework was the decomposition of the goal of a general task into a sequence of acts. As the granularity of these acts is dependent on the view of the domain expert, DACRON does not enforce a decomposition hierarchy but lets the users follow their own decomposition path. The users give the sequence of acts at a level they think is appropriate. The

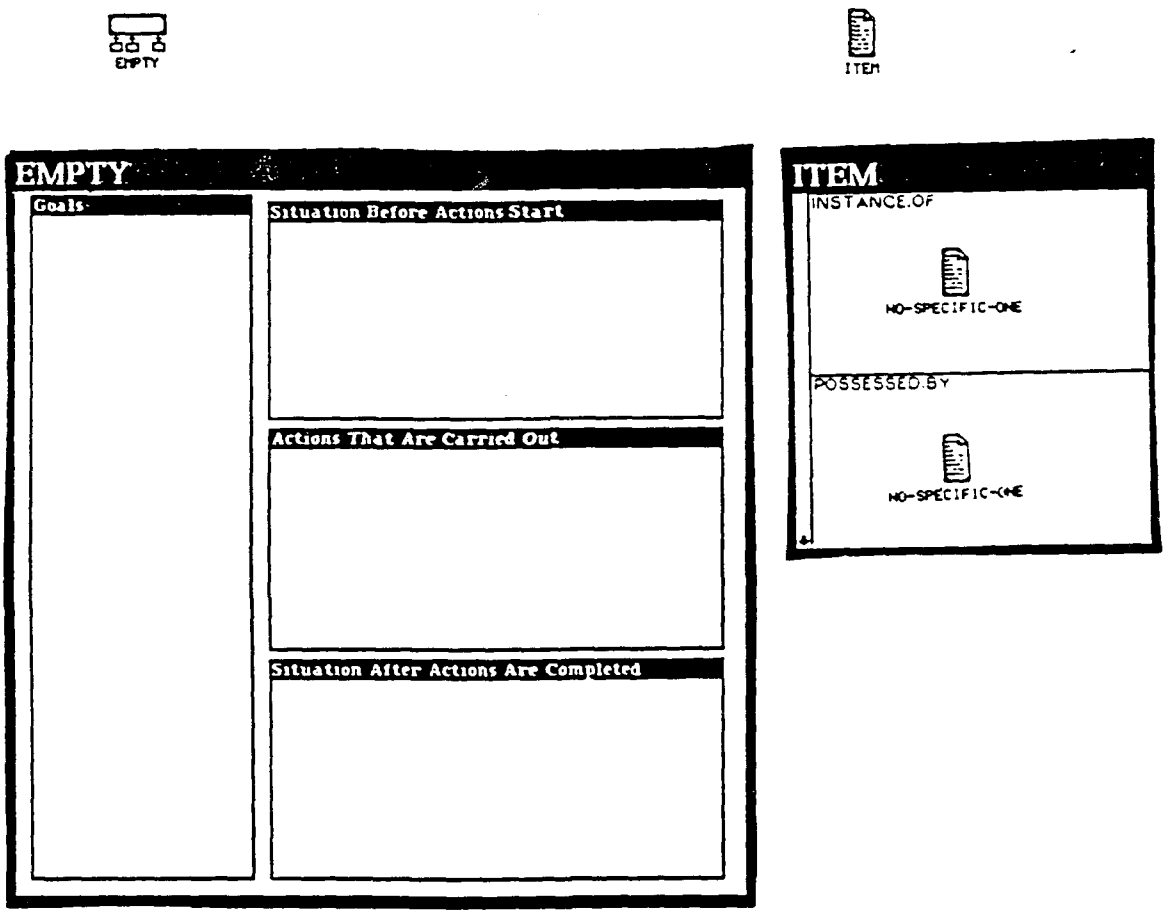

Figure 3: Icon and box representation for act type and object/relation type

users do this by specifying an act type. DACRON guarantees completeness of this specification by prompting the user for unspecified compartments.

Every item in the knowledge base is represented by an icon. A permanent window, called the *archive*, presents at any given moment a part of the knowledge base (figure 5). To see different parts of the knowledge base, the archive can be moved over the knowledge base, which is represented as a two-dimensional plane filled with icons. In addition, users may zoom in and out on the knowledge base.

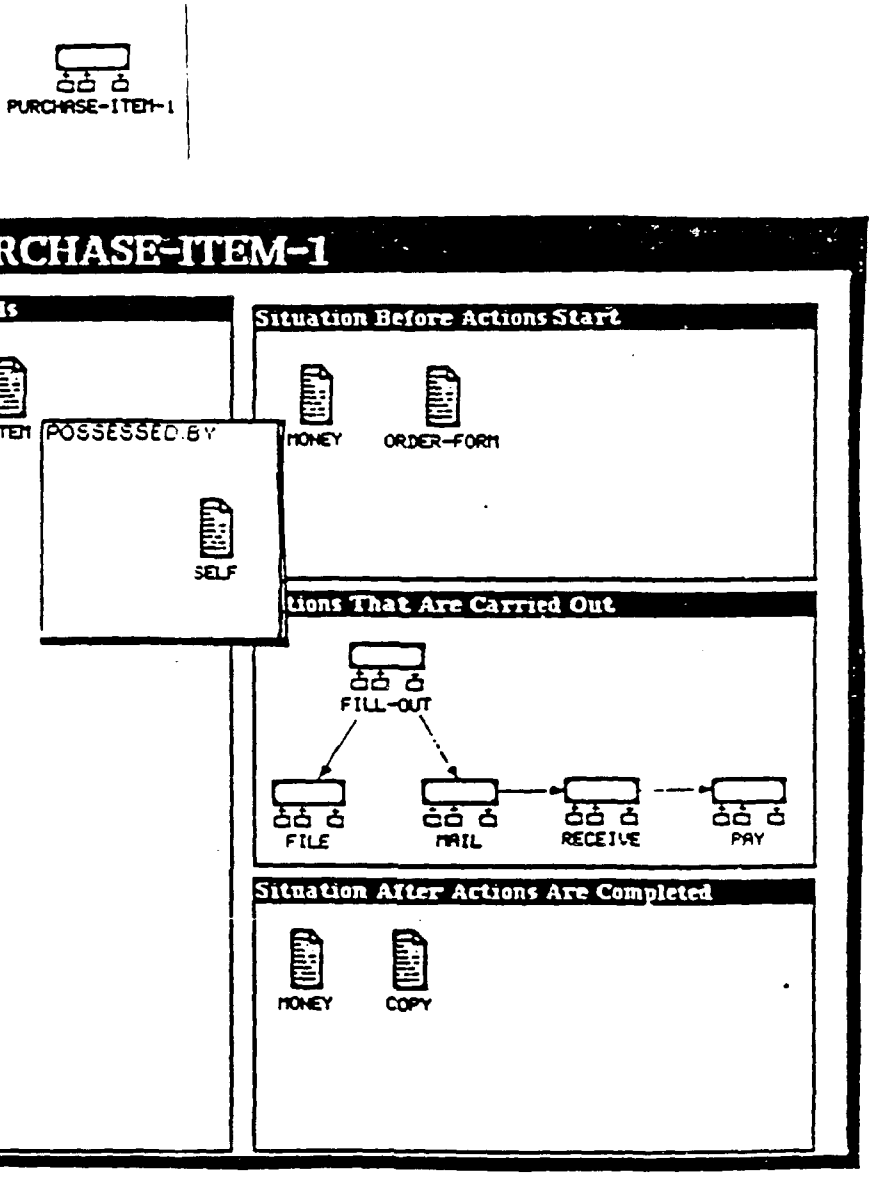

Figure 4: Icon and box for the purchase-task (with goal-value set).

To reorganize the clustering of icons in the knowledge base, users may employ the *retriever*. The retriever allows users to order the icons in the knowledge base by certain keys like name, date of creation, etc. or to search for icons by goal, constituent parts, etc. These retriever commands can be invoked by opening a special icon that is always present Upon opening this icon, users can pick commands from a menu and provide the desired options. The *help* icon is close to the *retriever* and provides general and specific help functions such as animated introduction to the system and help with special commands and tasks.

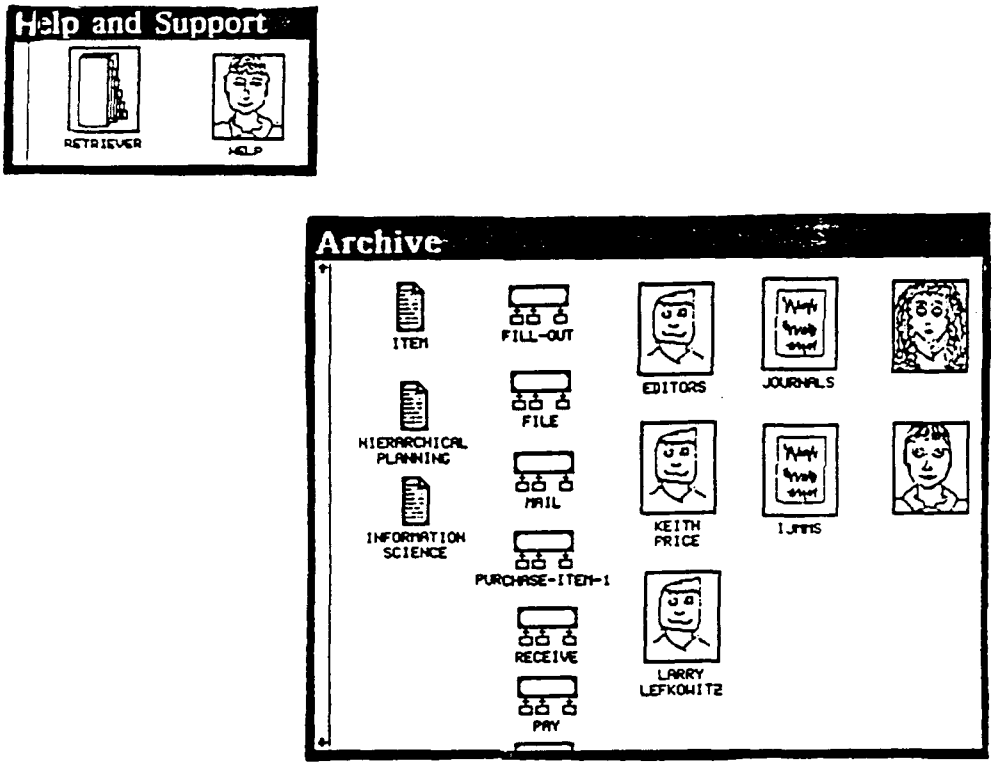

Figure 5: Archive

DACRON has two other facilities which are intended to aid users at the display stage and the debugging stage of the knowledge acquisition process. A *debugger* can be invoked to give an animated demonstration of the specification process of a type. The animation shows how the type was created, in which order arguments were given, what icons were placed into the type, etc. As this debugger is interactive, it is possible to stop it at any given moment and change the specification of the type shown. The animation can also be used for training purposes, teaching new users how DACRON is used.

The second facility is the *reviewer*. It is used to present sequences of plans and alternatives at crucial points in the planning process to the users. In this situation, we use the DACRON interface not only for the specification of activities, but also for the presentation of the planning process in POLYMER.

6.3.2.2.6 Future Work

We currently have the display components and rudementary acquisition components of the DACRON system implemented. We can show existing objects and activities in the DACRON-format. In the nearer future we will concentrate on the completeness of the acquisition component. Later work will be devoted to the animated presentation of planning processes and the use of animation in debugging the description of activities. We are also investigating the use of color as a coding technique for constraint definition.

We are planning to test and evaluate DACRON with users in a local office environment.

These user studies would involve the specification of office tasks by clerks and secretaries using DACRON and at a later point the display of whole office activities to office workers in the process of accomplishing complex tasks.

6.3.3 *Cooperative Problem Solving*

6.3.3.1 Planning and Execution of Tasks in Cooperative Work Environments

Problem solving has long been recognized as an important component in many work environments. The potentially complex and often cooperative nature of even apparently "routine" tasks has led to attempts to use planning techniques to support work in real-world domains. This chapter describes the work being done by the POLYMER project to construct a planning system to assist in the performance of multiagent, loosely-structured, underspecified tasks. Specifically, we present a representation for modeling tasks, agents and objects within such environments and describe the architecture and implementation of a planning system which uses these models to support cooperative work. We conclude with a description of how this planning system is being used to support further research in areas such as exception handling, negotiation and knowledge acquisition.

6.3.3.1.1 Introduction

Problem solving has long been recognized as an important component in many work environments [5, 33, 35]. The potentially complex nature of even apparently "routine" tasks has motivated the use of planning techniques to support work in real-world domains [18, 71]. However, the work in many environments is cooperative in nature. In such settings, tasks often cannot be performed by an individual; the coordinated effort of a group of people is needed to accomplish a desired goal. The size and complexity of certain tasks and the limited abilities, knowledge, skills and resources of any individual often make a cooperative approach the only way to achieve results. Offices, design teams, management structures, and factories are all examples of cooperative work environments.

As the scope and complexity of the problems addressed by computer-based support systems grows, the need for planning and knowledge-based approaches becomes more apparent. While traditional tools have been adequate to support single-person, small scale tasks (e.g., mail systems and forms tools in offices, compilers and debuggers in software development, etc.), supporting cooperative work requires the coordination of multiple agents using a variety of tools. By using a model of the tasks, objects and agents in an application domain to generate multi-agent plans, the POLYMER planning system [22, 23, 26] can coordinate interdependent activities in complex, underspecified domains.

Cooperative tasks are often "loosely structured" in that there may be a *typical* way (or ways) in which certain goals are accomplished, but the specifications are far from algorithmic. For instance, some steps within a task may be optional, there may be only a partial ordering of steps, any of several tasks may be used to achieve a goal, various agents may be able to perform a particular task, etc. The domain representation language and the associated planner must be able to capture and utilize any structure that is available but must also be able to cope with this flexibility.

This approach to supporting cooperative work addresses important research issues arising in several related fields, including distributed AI, the coordination of multiple agents (both cooperating and competing), and the attempt to reconcile strategic planning with situated actions [37, 39, 57, 80].

The use of planning to support cooperative work has just begun to be explored. The POLYMER project has focused on constructing a planning system to assist in the performance of

multiagent, loosely-structured, underspecified tasks. Such a system requires flexible models of the activities and objects in the application domain, and a means of using these models to help users achieve their goals.

In addition to serving as an aid to performing cooperative tasks, the POLYMER planner is being used as a testbed to support further research in the development of cooperative work environments (e.g., the SPANDEX, DACRON and GENEVA systems described in Section 6.3.3.1.4), and the development of advanced application domains (e.g., office automation environments, being developed both in our own research environment and by independent researchers).

In this chapter we present an overview of the POLYMER system, show its use in supporting cooperative work, and describe the current research centered around POLYMER. The following section describes the architecture and functionality of the POLYMER system. It presents POLYMER's model of an application in terms an example from the journal editing domain and shows how this model is used by the planner. Section 6.3.3.1.3 describes the planning process. It shows how a plan is interactively generated and what happens when problems arise. Section 6.3.3.1.4 presents a brief overview of research projects which are extending the POLYMER planner to develop an integrated cooperative work environment. Finally, the current status of the POLYMER project is summarized in Section 6.3.3.1.5.

6.3.3.1.2 The POLYMER system

Over the past two years, the POLYMER system has been developed as a testbed for supporting cooperative work. Using descriptions of domain tasks and objects, POLYMER combines strategic and reactive planning [80] and interacts with its users to generate a plan to accomplish a specified goal. POLYMER's domain description language evolved from one developed for the POISE intelligent interface system [25]. Using this formalism, POLYMER performs the type of hierarchical, non-linear planning described in NONLIN [81] and SIPE [84].

The POLYMER system has been developed using the KEE system[13] running on TI Explorers.[14] POLYMER uses KEE's frame-based knowledge representation to encode domain activity, agent and object descriptions (see Section 6.3.3.1.2.1). KEE's *assumption-based truth maintenance system* (ATMS) is used to record all planning decisions and support dependency-directed backtracking (see Section 6.3.3.1.3.3). We have extended KEE's "world" system to permit the construction of a world hierarchy graph to represent the state of the application domain at each point in a POLYMER plan (see Section 6.3.3.1.3.2). Finally, KEE was selected to obtain an added degree of portability, especially in the design of POLYMER's interface.[15]

The overall architecture of the POLYMER system is presented in Figure 1. The domain representation and planning methodology are presented below. POLYMER's exception handling capabilities and knowledge presentation and acquisition facilities are discussed briefly in Section 6.3.3.1.4.

[13] KEE is a registered trademark of IntelliCorp, Inc.

[14] Explorer is a trademark of Texas Instruments Inc.

[15] A version of POLYMER has recently been ported to Sun (trademark of Sun Microsystems, Inc.) workstations by Olivetti.

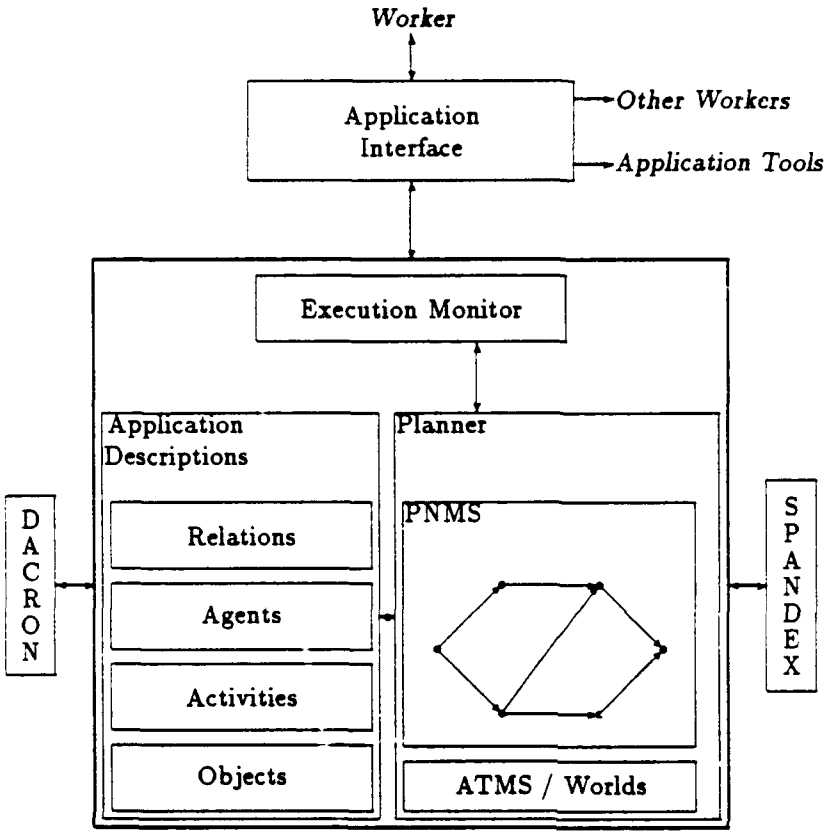

Figure 1: The POLYMER Planning System

6.3.3.1.2.1 Representing the application domain

POLYMER models an application domain though the description of *activities*, *objects* and *agents* within the domain. In this section we focus on the description of domain activities. A grammar for the POLYMER activities appears in Figure 2; a sample activity description from journal editing is shown in Figure 3.

The representation includes both the *goal* and *preconditions* of an activity as well as a *decomposition* of the activity into smaller steps. The steps are usually goals for which other activities will be selected during the planning process, but may also be specific activities or tool invocations (i.e., "actions"). Causal relations between steps may be specified and these relations are used to generate temporal ordering constraints as well as protection intervals.[16] Control flow among the steps (e.g., looping, iteration, etc.) may also be specified directly.

Consider an example from the domain of journal editing. When a paper is submitted for publication in a journal, the editor of that journal must select appropriate reviewers for the paper, invite them to review the paper, collect their reviews, make a decision based on the

[16]Protection intervals maintain a certain state of the world during a portion of the plan on the assumption that a state generated at one point will be needed at some later point. See [84].

activity	:=	**ACTIVITY** *activity-name* activity-clause*
activity-clause	:=	goal-clause # [preconditions-clause] # [effects-clause] # [decomposition-clause] # [rationale-clause] # [control-clause] # [agents-clause] # [constraints-clause]
goal-clause	:=	**GOAL** world-predicate
preconditions-clause	:=	**PRECONDITIONS** world-state*
effects-clause	:=	**EFFECTS** effect-spec*
decomposition-clause	:=	**DECOMPOSITION** step*
rationale-clause	:=	**PLAN-RATIONALE** enables-relation*
control-clause	:=	**CONTROL** control-construct*
agents-clause	:=	**AGENTS** agent-spec*
constraints-clause	:=	**CONSTRAINTS** world-state*
step	:=	step-spec [**done-by** agent-spec]
step-spec	:=	(**goal** *step-name* world-state) | (**activity** *step-name* {*activity-name* | (**one-of** *activity-name**)}) | (**action** *step-name action-name* parameter-list [world-state])
agent-spec	:=	world-state | kb-entity
parameter-list	:=	({variable, }* variable)
control-construct	:=	**before**{*step-name*, }+ *step-name* | **if** world-state **then** step-or-net [**else** step-or-net] | ***optional*** *step-or-net* | ***star*** *step-or-net* | **plus** step-or-net | **repeat** step-or-net repeat-bounds [**iterate-when** world-state]
repeat-bounds	:=	**while** world-state | **until** world-state | **times** *integer* | **for** variable **in** ({value, }* value) | **with** variable **suchthat** world-state
step-or-net	:=	*step-name* | (*step-name* **to** *step-name*)
enables-relation	:=	**enables**({*step-name*, }+ *step-name*)
effect-spec	:=	(effect-action world-state)
effect-action	:=	**set** | **add** | **delete**
world-state	:=	predicate (kb-term, kb-term) | **not**(world-state) | **and**({world-state, }* world-state) | **or**({world-state, }* world-state)
kb-term	:=	predicate (kb-term) | *kb-entity* | value | variable
predicate	:=	system-predicate | *kb-predicate*
system-predicate	:=	**member** | **subclass** | **equal**
value	:=	*string* | *number* | symbol-not-a-var
variable	:=	unconstrained-variable | constrained-variable
unconstrained-variable	:=	`any symbol whose first character is a "?"`
constrained-variable	:=	?(symbol-not-a-var {world-state | *kb-entity*})
symbol-not-a-var	:=	`any symbol whose first character is not a "?"`

Figure 2: A Grammar for Activities, Objects, and Agents

ACTIVITY: REVIEW-PAPER
Goal: reviews(?submission, ?reviews)
Preconditions:
member(?paper,papers)
edits.journal(?editor, ?journal)
submitted.to(?submisison, ?journal)
paper(?submission, ?paper)
Decomposition:
GOAL *reviewers-selected* =
and(reviewers(?submission, ?reviewers),
sufficient-reviewers(?journal, ?reviewers))
GOAL *paper-distributed* = has(?reviewer, *copy-of*(?paper))
GOAL *have-review* = has(?editor, review(?reviewer))
Plan Rationale:
ENABLES reviewers-selected paper-distributed
ENABLES paper-distributed have-review
Con-
trol: repeat (paper-distributed to have-review)
for ?reviewer **in** ?reviewers
Agents: ?(*editor*, editors)

Figure 3: A POLYMER Activity Description

reviews, and inform the author of the decision. Figure 3 shows a high-level POLYMER activity description for reviewing a submitted paper. It contains subgoals which are named and specified in terms of states of domain objects. The subgoals are causally (and therefore temporally) linked since a reviewer must be selected before a copy of the paper can be sent to the reviewer and the reviewer must receive the paper before the editor can get a review back. The two latter steps are repeated for each reviewer selected in the first step.

Using these domain descriptions, POLYMER interactively generates hierarchical, partially-ordered plans to accomplish a stated goal. The planning process, described below, is unique in its use of a combination of "script based" and "goal directed" descriptions of activities to overcome the rigidity of scripts while greatly reducing the cost of purely goal driven systems. It interleaves planning and plan execution in order to overcome the ambiguity inherent in complex, underspecified domains.

6.3.3.1.2.2 Preprocessing of domain descriptions

In order to make efficient use of the domain descriptions during the planning process, a certain amount of preprocessing is necessary. First, the external forms of the domain descriptions (shown above) are parsed and converted into Activity, Object, and Agent (KEE) units. Then, the activity descriptions are further processed to convert their information into a form more conveniently used during planning.

As we will see in the following section, POLYMER represents a plan as a *partially ordered network of plan nodes* called a *plan network*. To simplify the generation (and expansion) of

this plan network, POLYMER creates a plan network to represent each activity during the preprocessing phase. Thus, all the information in an activity description is converted into a partially ordered set of plan nodes. A more formal description of POLYMER's Plan Network Maintenance System appears in [7].

To generate a plan network for an activity, POLYMER first creates a plan node for each *step* in the activity's decomposition. *Goal*, *activity* and *action nodes* are generated for each corresponding step type. In addition, *structural* nodes are generated to represent the start and finish of the entire activity. Next, a partial ordering is established among the nodes (using the node's "predecessor" and "successor" fields) based on information in the activity's control clause. The more complex control constraints (e.g., *if*, *optional*, *star*, *plus* and *repeat*) result in the insertion of additional structural nodes.

Next, the causal relations in the plan rational clause are used to generate additional ordering constraints and protection intervals for the plan network. Finally, the activity's preconditions are transformed into constraints on the network's start-node, the activity's effects are placed on the network's finish-node, and any additional constraints specified in the activity description are placed on appropriate nodes within the network.

An example of the plan network generated for the activity description in Figure 3 is shown in Figure 4. A grammar for the specification of POLYMER plan networks and plan nodes appears in Figure 5.

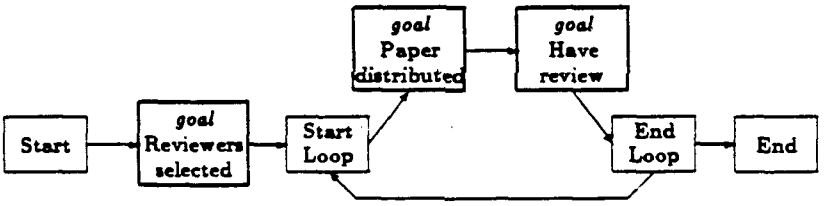

Figure 4: The "Referee Paper" Plan Network

In order to reduce the search for an activity to accomplish a goal during the planning process, one further preprocessing action is taken. For each goal node within an activity's plan network, POLYMER compares the value of the goal node to the goal of each known activity description. By finding and recording (during preprocessing) the set of all activities which could possibly satisfy each goal node, the planning process is made more efficient.

6.3.3.1.3 Interactive plan generation

To help a user achieve a desired goal, POLYMER attempts to generate a plan to accomplish the goal. To accomplish this, the goal and the required parameters must first be presented to the planner which then constructs the plan in a hierarchical manner. The planning proceeds in a top-down (in terms of plan abstraction), left-to-right (in terms of step ordering) fashion as far as possible without ambiguity. When the planning cannot proceed with certainty, it attempts to resolve the ambiguity by either 1) executing an action node (if any are "ready" as explained below) or 2) obtaining information from the user (such as which of several possible tasks it

```
plan-network        :=  PLAN-NETWORK network-name
                        start-node # finish-node # from-activity
start-node          :=  (start plan-node)
finish-node         :=  (finish plan-node)
from-activity       :=  (from-activity activity-name)
plan-node           :=  PLAN-NODE node-name node-info
node-info           :=  goal-node | activity-node | action-node | structural-node
goal-node           :=  generic-node # (type goal) # (goal world-state) #
                        (possible-activities activities-and-bindings) #
                        (selected-activity activity-name) #
                        (expansion-network plan-network)
activity-node       :=  generic-node # (type activity) # (goal world-state) #
                        (selected-activity activity-name)
action-node         :=  generic-node # (type action) # (goal [world-state]) #
                        (code fn-name)
structural-node     :=  generic-node # (type structural) # (loop-info loop-controller)
generic-node        :=  (predecessors (node-name*)) # (successors (node-name*)) #
                        (from-node node-name) #(from-activity activity-name) #
                        (before-world world-name) #(after-world world-name) #
                        (start-time time-range) # (finish-time time-range) #
                        (split-type {and | or}) #(join-type {and | or}) #
                        (level number) # (step-name step-name) # (agent agent-spec) #
                        (status { unseen | expanded | pending | phantom |
                                  complete | looping }) #
                        (repeat-from (node-name*)) #(repeat-after (node-name*)) #
                        (conditions world-state*) #(if-bound-conditions world-state*) #
                        (effects effect-spec*)
activities-and-bindings :=  (activity-and-bindings*)
activity-and-bindings   :=  (activity-name bindings-list)
bindings-list       :=  ((variable kb-term)*)
loop-controller     :=  LOOP-CONTROLLER controller-name loop-controller-info
loop-controller-info :=  (initial-list (kb-term*)) # (initial-value kb-term) #
                         (current-list (kb-term*)) # (current-value kb-term) #
                         (variable variable)
time-range          :=  (time-spec, time-spec)
time-spec           :=  number [time-unit]
time-unit           :=  seconds | minutes | hours | days | weeks | years
```

Figure 5: A Grammar for Plan Networks and Nodes

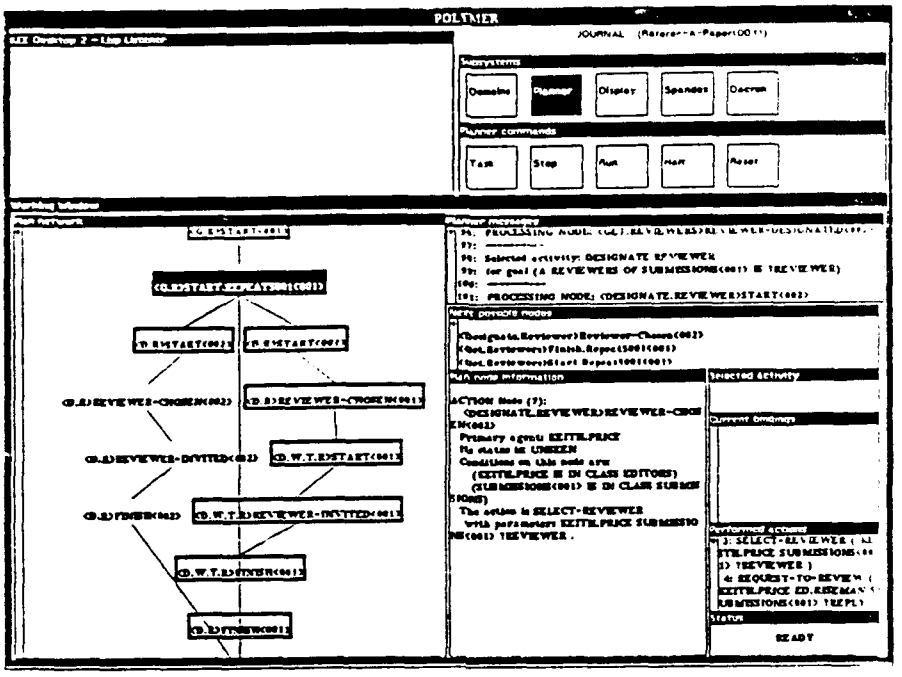

Figure 6: The POLYMER Developer's Interface

should use to accomplish an outstanding goal). In this section we describe how an initial goal is presented to the planner, how a plan network is interactively constructed to satisfy this goal, and what happens when difficulties arise during the generation and execution of the plan.

6.3.3.1.3.1 Statement of goal via an application interface

In a given application domain, there will typically be a fairly common set of goals that a particular user wishes to accomplish. For instance, in the journal editing domain, the editor of a journal will want to generate a request for papers, referee a submitted paper, modify a database of potential reviewers, etc. A reviewer will choose whether to review a particular paper and submit reviews to the editor. An author will write papers, submit them to journals, and rewrite the papers as necessary.

For a particular class of users in a particular domain, the user interface must allow the user to select the goal they want to accomplish and to specify the necessary parameters. Thus, a journal editor's interface allows the editor to state that he wishes to referee a paper and to specify the paper. The current system contains both a menu-driven "developer's" interface (Figure 6) and an iconic "end-user's" interface for the office domain (Figure 7).

Because each goal may require an extended period of time to accomplish, a user will most likely want to interleave several tasks. Thus, the interface allows the user to suspend the current goal and select another (new or previously suspended) one.

Once a goal is selected, POLYMER generates a top-level plan network to represent that goal. It consists of a single goal node contained between a pair of "start" and "finish" structural nodes. The user's parameterized goal is used as the goal-value of the goal node; any initial state information is added to the top-level start-node. Planning begins by expanding this plan network as explained below.

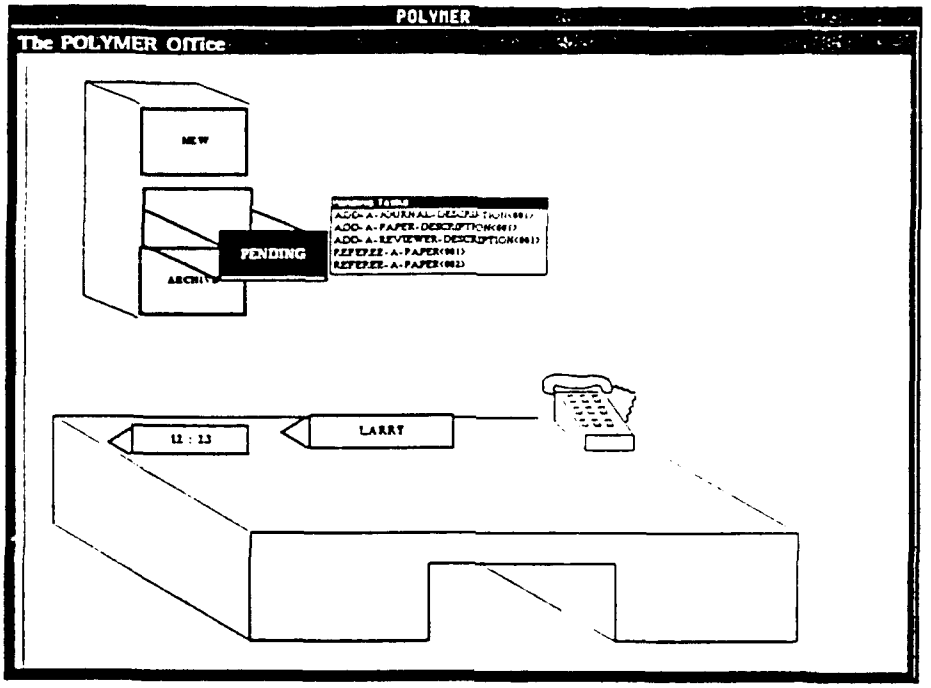

Figure 7: An Office Worker's Interface

6.3.3.1.3.2 Expanding the plan network

The POLYMER planner *hierarchically constructs* a plan by expanding a plan network. This expansion is performed by alternately selecting a plan node to process and then processing that node. The way in which a node is processed is determined by the node type.

Selecting a plan node

Because POLYMER constructs its plans in a top-down, left-to-right fashion, it selects the highest level (i.e., most abstract) node that is "ready" to be processed. In order to assure left-to-right processing, a node is "ready" to be processed if and only if 1) the node has not already been processed, 2) all of a node's necessary predecessors are complete and awaiting successors, and 3) all of the node's conditions are satisfied. A predecessor node is complete if its processing has been completed or if it does not need to be processed (i.e., a phantom goal node, as explained below).[17] A node's conditions are evaluated in the node's "before-world"[18] to assure that the node is applicable at this point in the plan.

To ensure top-down processing, each node is assigned an abstraction level denoting its "depth" from the original top-level goal. Thus, the nodes in the top-level network are assigned a level of 0; when a goal node is replaced by an activity network (as explained below), the level of the new

[17] A node's *join-type* determines whether it is necessary for all of the node's predecessors (for an "and" join) or merely any of them (for an "or" join) to be complete in order for the node to be "ready." Similarly, a node's *split-type* determines whether it is awaiting a successor: A node is awaiting successors until all of its successors are processed if it is an "and" split, or until any of its successors are processed if it is an "or" split.

[18] The "world" corresponding to the point in the plan immediately before the plan node is called its *before-world*; its *after-world* occurs immediately after the node.

nodes is 1 greater than the goal node they replace.[19]

Once the "ready" nodes are sorted by abstraction level, they are ordered by node type. Structural nodes (except for loop-iteration nodes) are processed first, followed in order by goal nodes, activity nodes, action nodes and finally loop-iteration nodes. If more than one node of a given type are ready to be processed, the planner can either select one based on a (modifiable) set of heuristics (e.g., specificity of the node's conditions, a priori preferences among tasks, etc.) or ask the user to select which node to process next.

Processing a plan node

Once POLYMER selects a plan node, the way in which the node is processed depends upon its type. For instance, it checks whether goal nodes are already satisfied by evaluating the goal node's value in the node's before-world. If the goal is already satisfied (either accomplished by some earlier steps or true in the initial world state), the node status is set to "phantom" and no further processing of the node takes place. If the goal is not satisfied, POLYMER determines if it can add additional ordering constraints to the existing plan network in order to place the node where its goal will be satisfied. If it is unable to achieve this, the planner must select an activity to achieve the goal.

As described in Section 6.3.3.1.2.2 above, each goal node template contains a list of all activities which may possibly satisfy the goal. These activities are now checked to see if their goals match that of the *instantiated* goal node and whether their preconditions are satisfied in the goal node's before-world. If more than one activity still qualifies as a means of accomplishing the goal, the planner can select one heuristically (using, for example, the closeness of the match between the activity's goal and that of the goal node, the specificity of the activity's preconditions, etc.) or by asking the user which activity should be performed.

Once an activity is selected for a goal, the plan network for that activity is instantiated and spliced into the current plan network in place of the goal node. Instantiating the network includes the instantiation of each of the nodes in the network, creation of KEE worlds corresponding to the new nodes, assertion of each node's effects in its after-world, and instantiating any needed protection intervals on these worlds. In addition to splicing the new nodes into the existing plan network, the new worlds are spliced into the existing world hierarchy.

Activity nodes are processed in a similar way, except that the selection of an appropriate activity is obviated. Action nodes require the invocation of tools (or interactions with other agents) and are handled by the *execution monitor*. The tools are invoked as specified in the "code" portion of the action node and the results are recorded in the after-world corresponding to the action node. Note that action nodes can be processed before the plan network is completely expanded. This permits the interleaving of planning and plan execution in order to prevent the planner from becoming swamped by the potentially explosive combinatorics of purely strategic planning in an underspecified and often ambiguous environment.

Most structural nodes simply serve as a means of demarcating activity boundaries and are the appropriate locations to place constraints and effects that belong at the beginning or end of an activity. Thus, an activity's *preconditions* and *effects* are placed on the activity's start and finish-nodes, respectively. The processing of these nodes only requires checking that their conditions are valid in order for them to be "complete." Structural nodes used to control looping, generated

[19]Therefore, nodes which are "higher" in terms of abstraction have "lower" level numbers.

from certain control constructs, are more complex. For each loop construct, a *loop-controller* object is created and the loop-iteration and loop-termination nodes have conditions and effects which utilize the loop-controller. Thus, if the conditions of a loop-iteration node are satisfied, an additional instantiation of the nodes which comprise the body of the loop are created and inserted into the plan network. If a loop-termination node is satisfied, both the loop-iteration node and the loop-termination node are marked "complete" and the looping terminates.

6.3.3.1.3.3 Detection and correction of plan problems

The expansion of the plan network and the execution of actions interleaved with plan generation can both cause problems to arise in the plan. These may be detected as 1) actions performed which were not expected as part of the plan, 2) goals selected by a user which were not currently expected, or 3) inconsistencies in the world model arising from violated protection intervals or failed constraints.

These problems can indicate an error on the part of the planner, "exceptional" behavior by the user, or simply a user error. The handling of exceptional behavior (i.e., intentional actions by the user that are not covered by the planner's domain model), is described in Section 6.3.3.1.4.1. In order to correct errors in the existing plan, POLYMER first considers the addition of node ordering constraints to resolve the problem. If this fails and the problem was caused by a violated goal, it considers reinstating the goal at a point after it was violated and before it is needed.

If POLYMER is unable to resolve the problem by manipulating the existing plan network, it is forced to backtrack and undo some of its previous planning decisions (e.g., the selection of an activity for a goal, a choice of alternative goals, etc.). Because POLYMER uses KEE's ATMS to justify and record each of its planning decisions, the planner needs to redo only those portions of the plan that led to the problematic results (i.e., dependency-directed backtracking).

6.3.3.1.4 Beyond the basic planner

While we believe that a planner such as POLYMER is an essential component to support work in cooperative environments, we also see the planner as the core of a set of sophisticated support tools. Several research projects are currently underway that build upon POLYMER's functionality and aim to extend its overall utility. These projects, described below, include systems to handle exceptional behavior, to present and acquire models of application domains, and to support conflict resolution.

6.3.3.1.4.1 Exception handling

Because POLYMER's domain model is inherently incomplete (as is *any* model of a "real-world" domain), there will be situations where the planner does not correctly anticipate a user's desired action(s). By combining the domain model with heuristic knowledge about how plans may deviate, the SPANDEX system [10] uses a process of *plausible inference* to generate *explanations* of how a user's exceptional behavior can be reconciled with an existing POLYMER plan. Using these explanations, SPANDEX constructs the necessary *amendments* to the domain model to incorporate this new behavior.

6.3.3.1.4.2 Knowledge presentation and acquisition

Though POLYMER's domain model may never be complete (or even necessarily correct), the ability to make modifications (e.g., additions, corrections) in a simple fashion is extremely important. In order to make such modifications *by end users* feasible, the domain model must be presented to and manipulated by the users in an "understandable" fashion. The DACRON project [55] has investigated how typical end-users perceive tasks and objects within their domains and has been able to map this more "natural" model onto the POLYMER formalism. An interactive, animated, iconic interface is being developed to permit the presentation of information to these users and allow them to modify existing information as well as to specify additional tasks, objects, etc.

6.3.3.1.4.3 Conflict resolution

Because cooperative work environments require the interaction of multiple agents (as well as the planner), conflicts between agents will inevitably arise. Differing goals, limited knowledge about the domain, varying capabilities and incomplete models of other workers can lead one agent to perform in a way that another agent does not expect. The *resolution* of such conflicts is often difficult and usually occurs though negotiation between the affected agents [6]. The GENEVA project [24] has begun to explore ways in which POLYMER's models of the domain and of the current plan can be used to *support* the negotiation process. In particular, it aims to assist in 1) initiating negotiation (by identifying the needed agents and presenting them with an appropriate view of the conflict), 2) maintaining the state of current and past negotiation sessions, 3) suggesting and allowing the exploration of solutions, and 4) *verifying that proposed* solutions actually resolve the conflict.

6.3.3.1.5 Summary

The POLYMER planning system has been designed and implemented as the core of an environment to support cooperative work. The current prototype has been used to interactively generate plans in such diverse areas as journal editing, software development and house purchasing. A preliminary version of the system has been delivered to an Olivetti research laboratory where it is being used to develop advanced applications in the area of office automation.

In addition to the development of further applications, POLYMER is serving as a testbed for several research projects. These projects are exploring the use of knowledge acquisition, exception handling, and computer-mediated conflict resolution as part of an effort to develop an integrated environment for the support of cooperative work.

6.3.4 *Tutoring Systems*

Representing, Acquiring, and Reasoning about Discourse Knowledge This five-year project was concerned with understanding human/machine communication in terms of the discourse actions, user modeling, and tutoring strategies. We have developed architectures and tools which facilitate the representation and acquisition of such knowledge. The tools have been incorporated into a generic and consistent foundation which has enabled us to represent, acquire, and reason about discourse and tutoring across several domains and from within several sites. Our goal is to enhance this framework and ultimately to produce a system in which "just plain folk," including psychologists, instructional scientists, teachers and domain experts, can work directly on the machine to modify and upgrade tutors without the need for knowledge engineers.

The big payoff has been that we can now apply the framework and evolving theory to several domains. We are not invested in promoting a particular discourse strategy, nor do we advocate a specific intelligent tutoring system design. Rather, we build tools that allow for a variety of system components, teaching styles, and intervention strategies to be combined into a single framework. For example, Socratic tutoring, incremental generalizations, and case-based reasoning are just a few of the teaching strategies we have experimented with using this formalism. Ultimately, we expect the machine to reason about its own choice of intervention method, to switch teaching strategies, and to use a variety of tactics and teaching approaches, while making decisions about the most efficacious method for managing one-on-one tutoring.

We are aided in our work by colleagues in three states who apply the tools we develop to new domains and new user groups.[20] For example, colleagues at San Francisco State University have sent us several carefully built physics simulations on top of which we placed the tutoring formalism described here.[21] These colleagues help us evaluate the tutors. Using an iterative methodology in which formative evaluation augments tutor development, we have designed systems that tutor about statics, thermodynamics, time management, statistics, genetics, algebra word problems, and explanations. In this section we describe that methodology along with the generic tutoring foundation.

Development of intelligent tutors, like development of any artificial intelligence system, requires several iterative cycles: computer scientists and instructional designers first collaborate on the design and development of the system, additional collaboration is required to test the system with students, and then the original implementation is modified and refined based on information gained through testing. This cycle is repeated as time permits.

For example, a professor at City College of San Francisco used the statics tutor (Section 6.3.5.1.1) in a classroom and noticed weaknesses in the simulation's ability to inform the student. She augmented the system with verbal discourse, adding examples or explanations, making diagnoses, and clarifying system response. She gave us a list of her additional discourse moves to be incorporated into the next version of the tutor.

[20] Participant institutions include San Francisco State University, San Francisco City College, Trinity College in Hartford, CT, and State University of New York at Plattsburgh, NY.

[21] San Francisco State University, the University of Massachusetts, and the University of Hawaii are members of the Exploring System Earth Consortium (ESE), a group of universities and industries working together to build intelligent science tutors. The consortium is supported by the Hewlett-Packard Corporation.

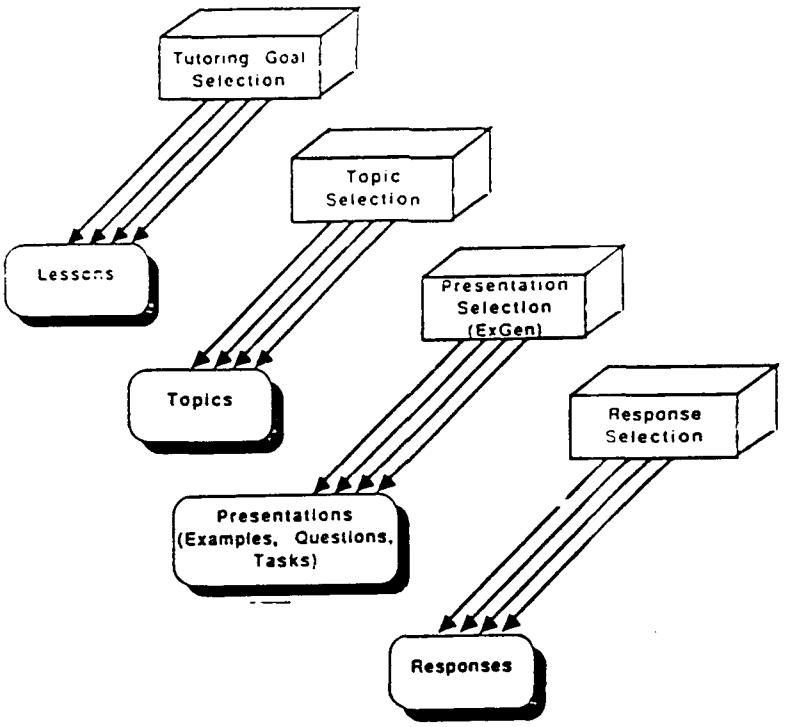

Figure 1: Representation and control in a tutoring system.

6.3.5 *Representation and Control*

Knowledge bases for human-machine communication might store concepts, activities, relations between topics, and other quantities needed to make expert decisions. For tutoring, they might store a variety of lessons, topics, presentations, and response selections available to the tutor (see Figure 1). The control structures might be specified at the four levels indicated in Figure 1, separately defining control for selection of lesson, topic, presentation, and response selection.

Currently, our control structures are motivated by specific instructional and diagnostic goals; thus, for example, one control structure produces a predominantly Socratic interaction and another produces interactions based on presenting incrementally generalized versions of new concepts or examples. Control structures are specific to a particular level of control and are used separately to define the reasoning to be used for selecting a lesson, topic, presentation, or response.

Acquiring and encoding this large amount of knowledge, or the knowledge acquisition process, is difficult and time consuming. We have built a number of tools that facilitate representating, acquiring, and reasoning about tutoring knowledge (see Figure 2). For each knowledge base (lessons, topics, presentation, or response) we consider the nature of the knowledge that must be accessed, such as the examples or questions (from the presentation knowledge base) or the activity the tutor must engage in, such as to motivate or teach a topic, or to provide follow-up. We have built tools, shown at the bottom of Figure 2, to support most activities listed in the figure. Only a few such tools will be described in this section, namely TUPITS, Exgen, Response Matrix, DACTN, and multiple views.

We divide the discussion into two parts, separately describing tools for representing tutoring primitives (lessons, topics, and presentations) and then tools for representing discourse knowl-

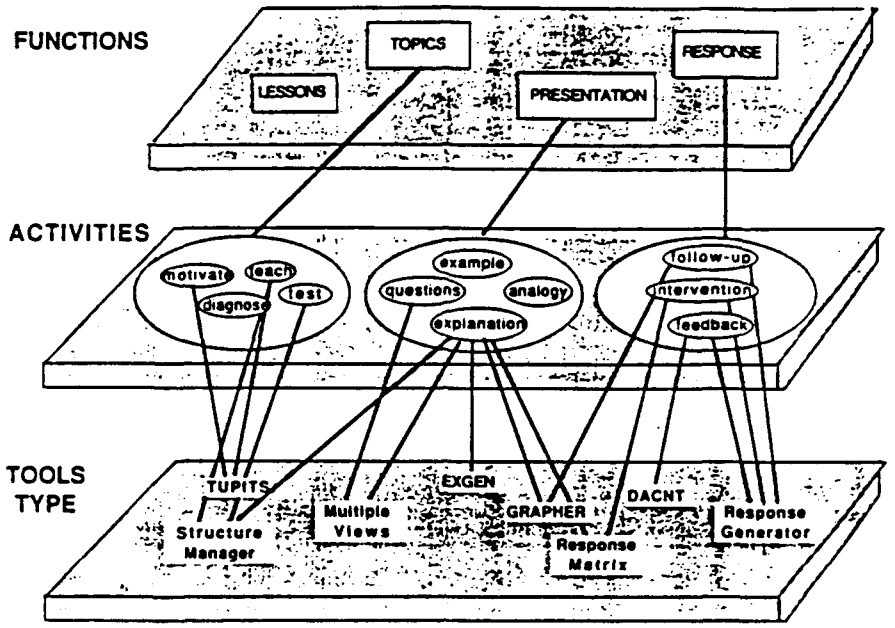

Figure 2: Tools for the representation and control of tutoring knowledge.

edge.

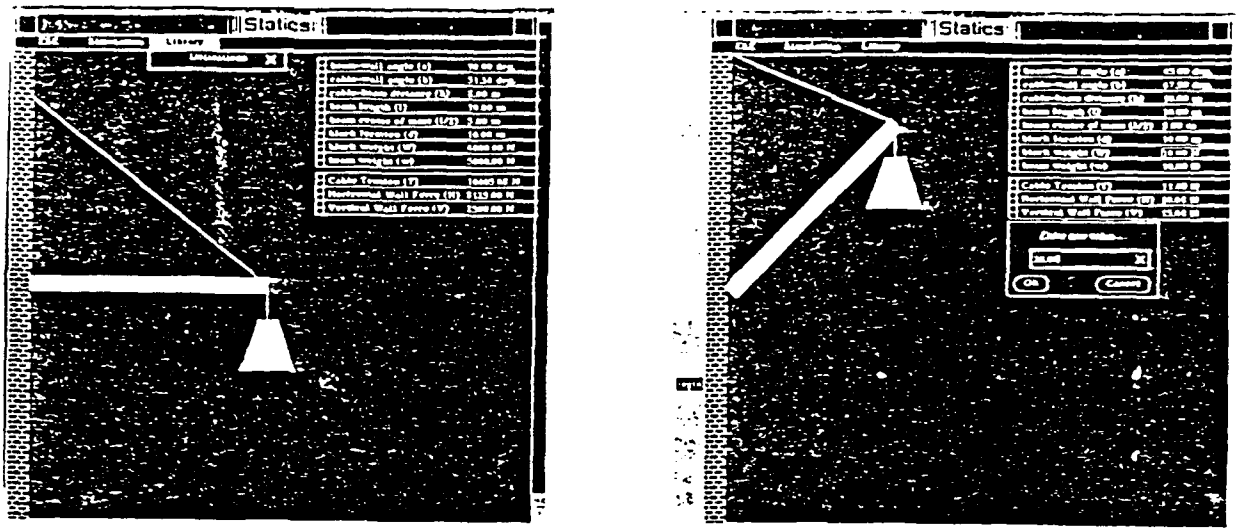

Figure 1: Statics tutor.

6.3.5.1 Tools for Representing Tutoring Primitives

We define tutoring primitives as basic elements needed for communicating knowledge, such as topics to be taught, specific tutoring responses, and possible student errors. Our knowledge bases hold a variety of examples, knowledge types, tasks to be given to the student, and discourse states describing various human-machine interactions.

6.3.5.1.1 Example Tutoring Primitives

As an example of how tutoring primitives are used, we describe two tutors we have built in conjunction with the Exploring Systems Earth (ESE) Consortium [29]. These tutors are based on interactive simulations that encourage students to work with "elements" of physics, such as mass, acceleration, and force. The goal is to help students generate hypotheses as necessary precursors to expanding their own intuitions. We want the simulations to encourage students to "listen to" their own scientific intuition and to make their own model of the physical world before an encoded tutor advises them about the accuracy of their choices. These tutors have been described elsewhere [87, 89] and will only be summarized here.

Figure 1 shows a simulation for teaching concepts in introductory statics. In this example, students are asked to identify forces and torques on the crane boom, or horizontal bar, and to use rubber banding to draw appropriate force vectors directly on the screen. When the beam is in static equilibrium there will be no net force or torque on any part of it. Students are asked to solve both qualitative and quantitative word problems.

If a student were to specify incorrect forces either by omitting force lines or by including the wrong ones, the tutor makes a decision about how to respond. There are many possible responses depending on the tutorial strategy in effect. The tutor might present an explanation or hint, provide another problem, or demonstrate that the student's analysis leads to a logical contradiction. Still another response would be to withhold explicit feedback concerning the

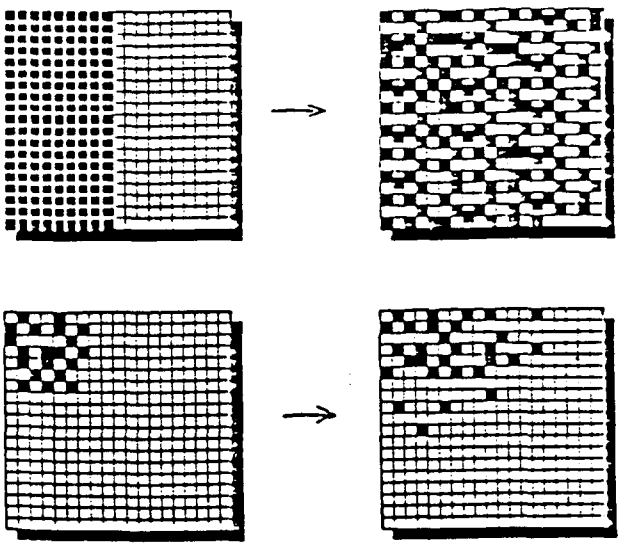

Figure 2: Thermodynamics tutor.

quality of the student's answer, and to instead demonstrate the consequence of omitting the "missing" force (i.e., the end of the beam next to the wall would crash down). Such a response would show the student how his/her conceptions might be in conflict with the observable world and to help him/her visualize both an internal conceptualization and the science theory.

A second tutor is designed to improve a student's intuition about concepts such as energy, energy density, entropy, and equilibrium in thermodynamics. It makes use of a very simplified but instructive simulated world consisting of a two-dimensional array of identical atoms (Figure 2; [4]. Like the statics tutor, the thermodynamics tutor monitors and advises students about their activities and provides examples, analogies, or explanations. In this simplified world the atoms have only one excited state; the excitation energy is transferred to neighboring atoms through random "collisions." Students can specify initial conditions, such as which atoms will be excited and which will remain in the ground state. They can observe the exchange of excitation energy between atoms, and can monitor, via graphs and meters, the flow of energy from one part of the system to another as the system moves toward equilibrium. In this way, several systems can be constructed, each with specific areas of excitation. For each system, regions can be defined and physical qualities, such as energy density or entropy, plotted as functions of time.

6.3.5.1.2 Representing and Reasoning about Tutoring Primitives

For each domain described above, topics, examples, explanations, and possible misconceptions are represented in the four knowledge bases described in Section 6.3.5. We use a network of Knowledge Units frames to explicitly express relationships between topics such as prerequisites, corequisites, and related misconceptions (Figure 3). An important notion about the network is that is declarative—it contains a structured space of concepts, but does not mandate any particular order for traversal of this space.

The network describes tutorial strategies in terms of a vocabulary of primitive discourse

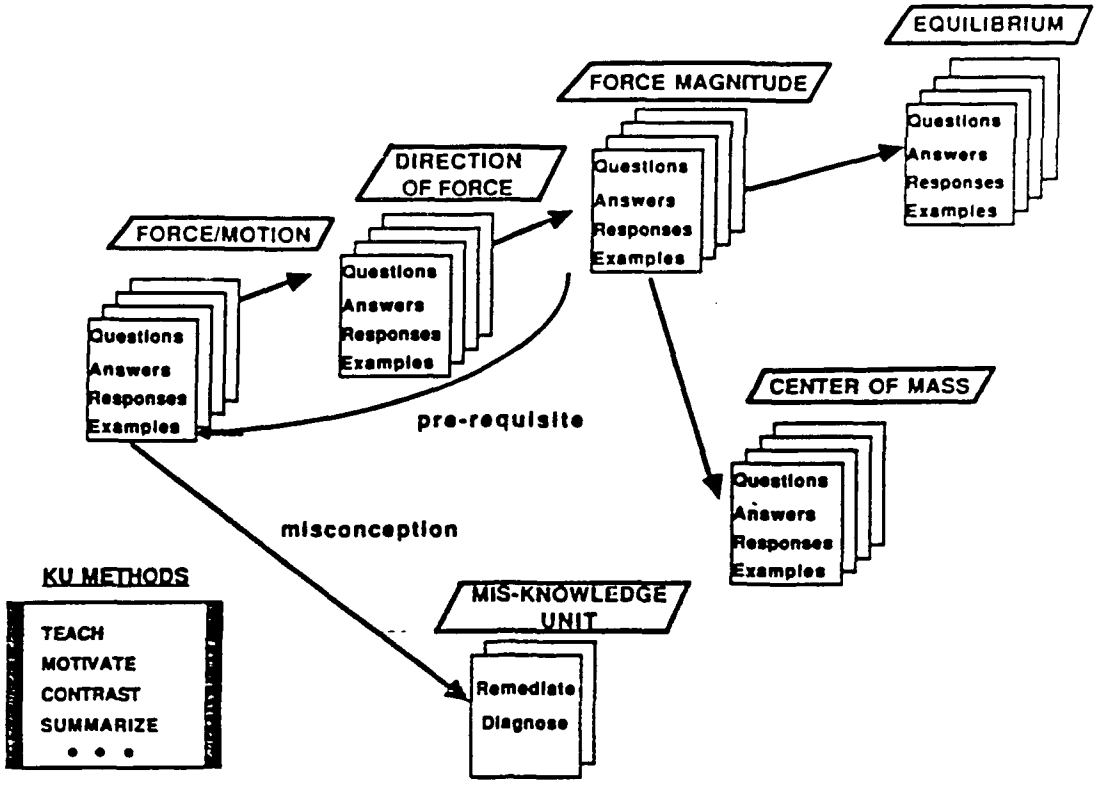

Figure 3: Hierarchy of frames.

moves such as teach, motivate, contrast, and summarize. It is implemented in a language called TUPITS[22] which was built as a framework to facilitate development of numerous tutors. It is an object-oriented representation language that provides a framework for defining primitive components of a tutorial discourse interaction. These components are then used by the tutor to reason about its next action.

As shown in Figure 3, each object in TUPITS is represented as a frame and each frame is linked with other frames representing prerequisites, corequisites, or triggered misconceptions. The primary objects in TUPITS are:

- Lessons which define high-level goals and constraints for each tutoring session;
- Knowledge Units (KUs);
- MIS-KUs, which represent common misconceptions, wrong facts or procedures, and other types of "buggy" knowledge;
- Examples, which specify parameters that configure an example, diagram, or simulation to be presented to the student;
- Questions, which define tasks for the student and how the student's behavior during the task might be evaluated; and
- Presentations, which bind an example together with associated questions.

[22]TUPITS (Tutorial discourse Primitives for Intelligent Tutoring Systems) was developed by Tom Murray and runs on both Hewlett-Packard Bobcats and Apple Mackintosh IIs.

MIS-KUs, or "Mis-Knowledge Units," represent common misconceptions or knowledge "bugs" and ways to remediate them. Remediation is inserted opportunistically into the discourse. The tutoring strategy parameterizes this aspect of Knowledge Unit selection by indicating whether such remediation should occur as soon as the misconception is suspected, or wait until the current Knowledge Unit has been completed.

Control is achieved through information associated with each object which allows the system to respond dynamically to new tutoring situations. For instance, Knowledge Units, or topics represented as objects, have procedural "methods" associated with them that:

- teach their own topic interactively;
- teach their own prerequisites;
- explain knowledge didactically;
- test students for knowledge of that topic;
- summarize themselves;
- provide examples of their knowledge (an instantiation of a procedure or concept);
- provide motivation for a student learning the topic; and
- compare this knowledge with that of other Knowledge Units.

A specific tutoring strategy manifests itself by parameterizing the algorithm used to traverse the knowledge primitives network based on classifications of and relations between knowledge units. Several major strategies have thus far been implemented. For example, the tutor might always teach prerequisites before teaching the goal topic. Alternatively, it might provide a diagnostic probe to see if the student knows a topic. Prerequisites might be presented if the student doesn't exhibit enough knowledge on the probe. These prerequisites may be reached in various ways, such as depth-first and breadth-first traversal. An intermediate strategy is to specialize the prerequisite relation into "hard" prerequisites, which are always covered before the goal topic, and "soft" prerequisites, taught only when the student displays a deficiency.

Control and Reasoning about Examples. Another example of reasoning about tutoring primitives is shown by the activities of ExGen [79, 88]. ExGen takes requests from various components of the tutor and produces an example, question, or description of the concept being taught. For example, the two configurations in Figure 2 and the two "universes" in Figure 4 can be produced by ExGen. A "seed" example base contains prototypical presentations of each type. ExGen's modification routine expands this into a much larger virtual space of presentations as needed. The goal is to enable the tutor to have flexibility in its presentation of examples and questions/tasks that accompany those examples, without needing to represent all possible presentations explicitly.

ExGen is driven by *example generation specialists*, or knowledge sources, each of which examines the current discourse and student model and produces requests (weighted constraints) to be given to ExGen. Example generation specialists may be thought of as tutoring rules, encoding such general prescriptives as "when starting a new topic, give a start-up example," or "ask questions requiring a qualitative response before those involving quantities."

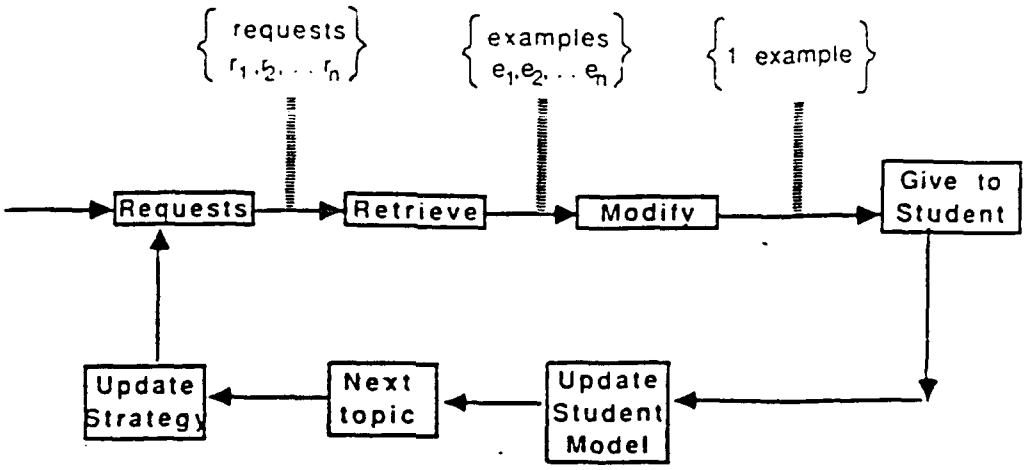

Figure 4: Reasoning about examples.

Requests input to ExGen are expressed as weighted constraints called *requests* (see Figure 4). The constraints are written in a language that describes logical combinations of the desired attributes of the example, and the weights on them represent the relative importance of each of these attributes. Attributes include boom angle or boom length for the statics tutor and universe size and density for the thermodynamics tutor. The returned example generally meets as many of the constraints as possible in the priority indicated by the weights.

The tutoring strategy impacts on this layer of presentation selection by prioritizing the relative importance of the recommendations produced by each of the example generation specialists. Within a strategy, each specialist has a weight multiplied by the weight of the requests produced by the specialists. Altering the behavior of the presentation control is simply a matter of changing the weights on the specialists by selecting a new strategy.

For instance, one specialist requests that presentations describing the current Knowledge Unit be given and another requests that the student be questioned. These competing requests are ordered by the current tutoring strategy. We are also examining strategies for temporal ordering of the presentation of examples, such as Bridging Analogies [13, 61] and Incremental Generalization.

Acquiring Tutoring Primitives Knowledge. Knowledge acquisition of tutoring primitives knowledge or acquiring and encoding the questions, examples, analogies, and explanations used in a particular domain is still a difficult problem. We need to know not only the primitives used by the expert, but also the reasoning he/she uses to decide how and when to use each primitive. We achieve knowledge acquisition for tutoring primitives through a graphical editor built into TUPITS which is used by the instructional designer to encode and modify both primitives and

the reasons why one primitive might be used over another. The graphical editor allows a teacher to generate and modify primitives without working in a programming language. The system currently presents a user with a sheaf of "cards" listing a series of primitives. The user chooses a card and brings the primitive into an edit window, from which he/she builds new primitives.

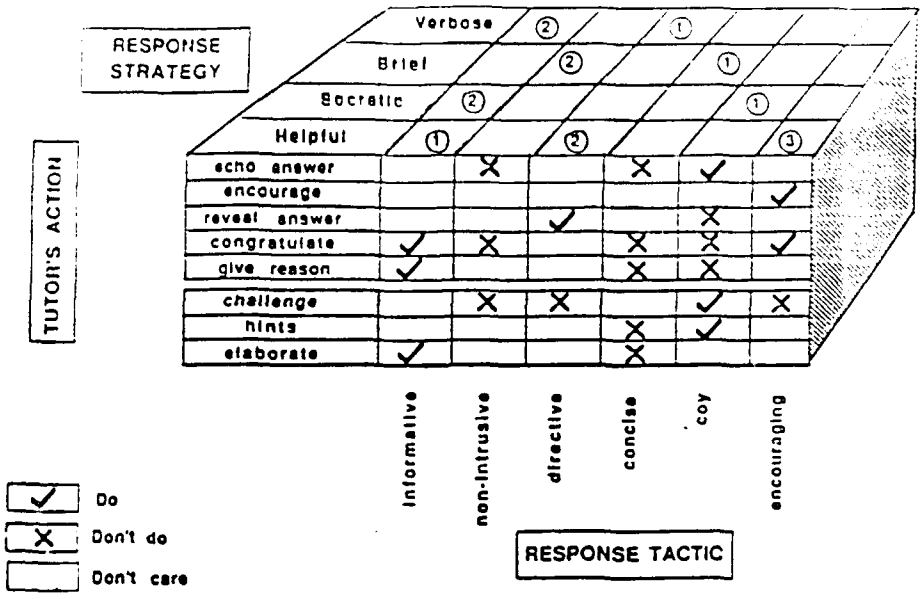

Figure 1: Reasoning about discourse.

6.3.5.2 Tools for Representing Discourse Knowledge

Our tutors are beginning to represent and reason about alternative responses to the student. Choices are concerned with how much information to give and what motivational comments to make. For instance, the machine must decide whether or not to:

- talk *about* the student's response;
- provide *motivational* feedback about the student's learning process;
- say *whether* an approach is appropriate, *what* a correct response would be, and *why* the student's response is correct or incorrect;
- provide hints, leading questions, or counter-suggestions.

Motivational feedback may include asking questions about the student's interest in continuing or providing encouragement, congratulations, challenges, and other statements with affective or prelocutionary content. Control is modulated by which tutoring strategy is in effect, which in turn places constraints on what feedback or follow-up response to generate. The strategy may also specify that system action be predicated on whether the student's response was correct, or whether any response was given.

Reasoning about Discourse Level. As a start to this process we have defined several high-level response strategies and tactics (see Figure 1). For example, we have designated an informative response tactic as one in which the machine will elaborate, give reasons, and congratulate the student. For each concept represented in the machine, some of these primitive responses are available and the machine will generate the requested tactic. However, we also advise the system about strategies such as Socratic tutoring, being brief, and being verbose. Here we indicate a priority ordering; thus to be Socratic, the machine must place highest priority on the tactic called coy and secondary rating on the tactic to be informative. If there is a conflict between the checks and the crosses in the model shown in Figure 1, that notation with the highest priority will win.

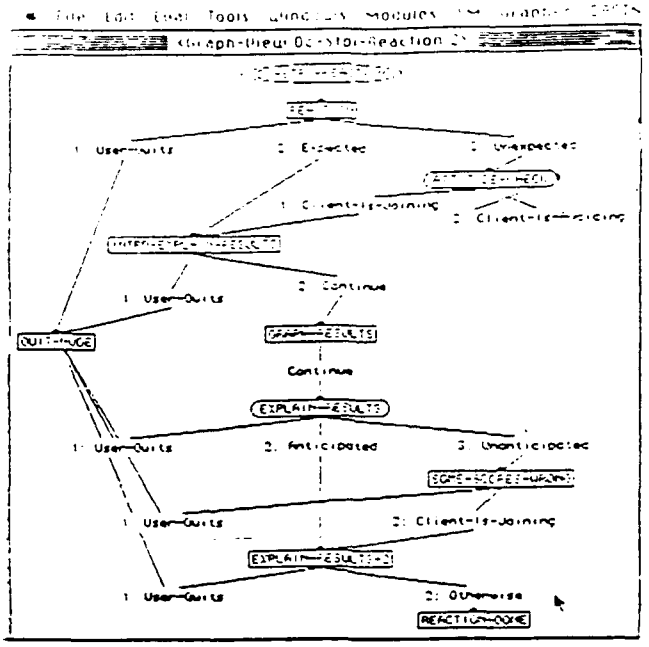

Figure 2: Discourse ACtion Transition Network: DACTN.

6.3.5.2.1 Managing Discourse

We realize that a more flexible and responsive discourse management technique is critical to a tutoring or consultant system. By discourse management, we mean the system's ability to maintain interactive discourse with the user and to custom-tailor its responses beyond the generalized discourse levels suggested above. Ideally, the system should tailor its response to the idiosyncrasies of a particular user. Machine discourse and response need not be in natural language to be effective [75].

For example, the system should ensure that an intervention relates directly to an individual's personal history, learning style, and on-line experience with the system. It should dynamically reason about a user's actions, the curriculum, and the discourse history. In doing this the tutor should make each user feel that his/her unique situation has been responded to appropriately and sensitively. In this way the system simulates one-on-one human tutoring behavior. The mechanism we use to do this is called a DACTN, *Discourse ACtion Transition Network*,[23] which represents and controls human-machine dialog. Figure 2 shows a DACTN for responding to a user about an inventory test of questions that he/she took in a consultant system called TEV, which advises people about setting priorities and using time effectively. This graphic is taken directly off the screen of that system.

Sometimes the intervention steps designated by a DACTN are based on a taxonomy of frequently observed discourse sequences which provide default responses for the tutor [89]. The discourse manager reasons about local context when making discourse decisions. Here local context is an aggregate of the client profile and response history.

The DACTN represents the space of possible discourse situations: Arcs track the state of the conversation and are defined as predicate sets while nodes provide actions for the tutor. The discourse manager first accesses the situation indicated by the arcs, resolving any conflicts

[23] Rhymes with ACT-IN.

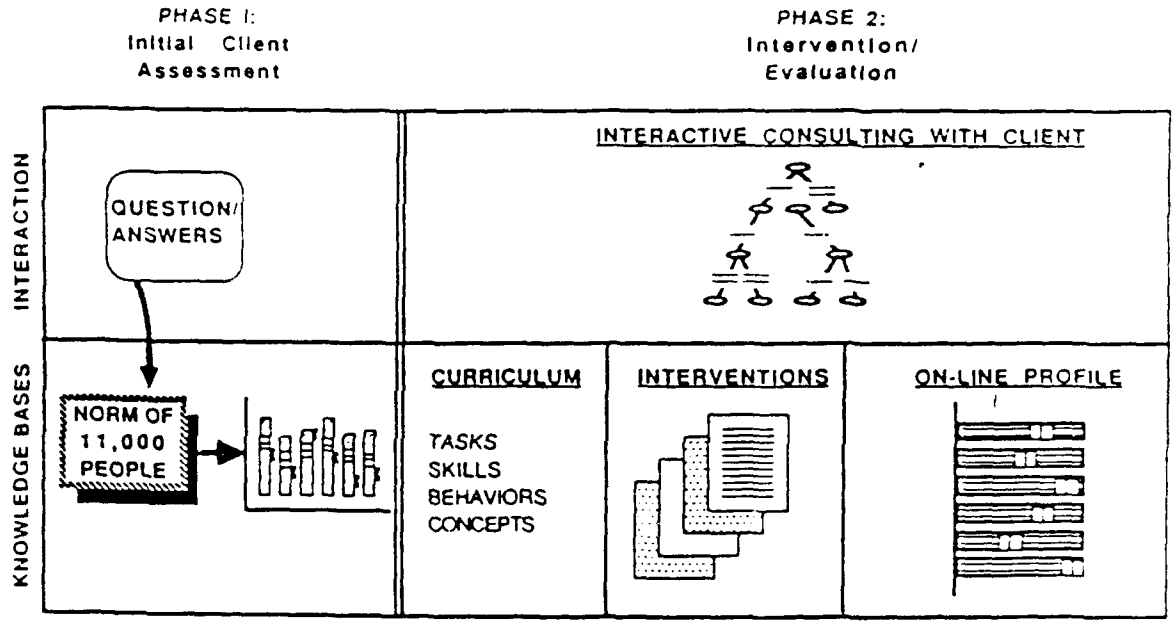

Figure 3: Two phases of the consultant.

between multiply-satisfied predicate sets, and then initiates the action indicated by the node at the termination of the satisfied arc.

Arcs represent discourse situations defined by sets of predicates over the client profile and the state of the system. For instance, the value of the arc "CLIENT-IS-AVOIDING" (top-half of Figure 2) is determined by inferring over the current state of the profile and recent client responses. Placing actions at the nodes rather than on the arcs, as was done in the ATN [86], allows nodes to represent abstract actions which can be expanded into concrete substeps when and if the node is reached during execution of the DACTN. For example, the node "EXPLAIN RESULTS" (middle of Figure 2) expands into yet another complete DACTN to be executed if this node is evaluated in the course of the intervention.

Each user response causes the user model, or in this case the personality profile, to be updated, which in turn affects the interpretation and resolutions of subsequent interactions. DACTNs allow discourse control decisions to be based on a dynamic interpretation of the situation. In this way the mechanism remains flexible, domain-independent, and able to be dynamically rebuilt—decision points and machine actions are modifiable through a graphics editor. DACTNs have been implemented in two domains, one which fine-tunes explanations for specific users and discourse history and one which trains people to improve their time management skills.

Bibliography

[1] Ambros-Ingerson, J.A., Steel, S. "Integrating Planning, Execution, and Monitoring," *Proceedings of AAAI-88*, Minneapolis-St. Paul, Minnesota, pp. 83-88.

[2] Anderson, J.R., "Architecture of Cognition," Harvard University Press, 1983.

[3] Anderson, J.R. and Bower, G.H., "Human Associative Memory," Winston and Sons, 1973.

[4] Atkins, T., *The second law*, San Francisco: Freedman, 1982.

[5] Barber, G.R., "Supporting Organizational Problem Solving with a Work Station," *ACM Transactions on Office Information Systems*, 1, pp. 45–67, 1983.

[6] Bartos, O.J., "Process and Outcome of Negotiation," Columbia University Press, New York, 1974.

[7] Beetz, M. and Lefkowitz, L.S., "Reasoning about Justified Events: A Unified Treatment of Temporal Projection, Planning Rationale and Domain Constraints," *Technical Report CSL-89-6*, Collaborative Systems Laboratory, Computer and Information Science Department, University of Massachusetts, Amherst MA, 1989.

[8] Boose, J., "Personal construct theory and the transfer of human expertise," *Proceedings of the national conference on artificial intelligence*, Austin, Texas, 1984.

[9] Broverman, C.A. and Croft, W.B. "Plausible Explanations to Cope with Unanticipated Behavior in Planning," COINS Technical Report 88-56, University of Massachusetts, Amherst, Ma. June 1988.

[10] Broverman, C.A. and Croft, W.B., "Reasoning about Exceptions during Plan Execution Monitoring," *Proceedings of the AAAI-87*, 1987.

[11] Broverman, C.A. and Croft, W.B. "SPANDEX: An Approach Toward Exception Handling in an Interactive Planning System," COINS Technical Report 87-127, University of Massachusetts, Amherst, Ma. December 1987.

[12] Broverman, C.A., and Croft, W.B. "Exception Handling During Plan Execution Monitoring," *Proceedings of AAAI-87*, July 1987, Seattle, WA.

[13] Brown, D., Clement, J. & Murray, T., "Tutoring specifications for a computer program which uses analogies to teach mechanics," *Cognitive Processes Research Group Working Paper*, Department of Physics, University of Massachusetts, Amherst, MA, 1986.

[14] Card, S.K., Moran, T.P. and Newell, A., *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum, 1983.

[15] Carnegie Group Inc. "Knowledge Craft Users Manual," 1986.

[16] Carver, Norman, *Evidence-Based Plan Recognition*, TR 88-13, Computer and Information Science Department, University of Massachusetts, 1988.

[17] Carver, N., Lesser, V.R., and McCue, D., "Focusing in Plan Recognition," *Proceedings of AAAI-84*, 1984, 42–48.

[18] Chapman, D., "Planning for Conjunctive Goals," *Artificial Intelligence*, 32, pp. 333–377, 1987.

[19] Clancey, William, "From GUIDON to NEOMYCIN and HERACLES in Twenty Short Lessons," *AI Magazine*, 7 (3) 1986, 40–60.

[20] Clancey, William, "Classification Problem Solving," *Proceedings of AAAI-84*, 1984, 49–55.

[21] Cohen, Paul, *Heuristic Reasoning About Uncertainty: An Artificial Intelligence Approach*, Pitman, 1985.

[22] Croft, W.B. and Lefkowitz, L.S., "A Goal-based Representation of Office Work," *Proceedings of the IFIP Conference on Office Knowledge*, 1988. (Also in *Office Knowledge: Representation, Management, and Utilization*, North Holland, 1988.)

[23] Croft, W.B. and Lefkowitz, L.S., "Knowledge-based Support of Cooperative Activities," *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988. (Also in *Readings on Distributed AI*, Morgan Kaufmann, 1988.)

[24] Croft, W.B. and Lefkowitz, L.S., "Computer-Mediated Conflict Resolution," Technical Report, COINS Department, University of Massachusetts, Amherst, Massachusetts, May 1988.

[25] Croft, W.B. and Lefkowitz, L.S., "Task Support in an Office System," *ACM Transactions on Office Information Systems*, Vol. 2, July 1984.

[26] Croft, W.B. and Lefkowitz, L.S., "Using a Planner to Support Office Work," *Proceedings of the ACM Conference on Office Information Systems*, March 1988.

[27] Croft, W.B., Lefkowitz, L.S., Lesser, V.R. and Huff, K., "POISE: An Intelligent Assistant for Profession Based Systems," *Proceedings of the Conference on Artificial Intelligence*, Oakland University, Michigan, 1982.

[28] Doyle, J., "A Truth Maintenance System," *Artificial Intelligence*, 12(1979), 231-272.

[29] Duckworth, W., Kelley, J., & Wilson, S., "AI goes to school," *Academic Computing*, 1987.

[30] Durfee, Edmund, and Victor Lesser, "Incremental Planning to Control a Time-Constrained, Blackboard-Based Problem Solver," *IEEE Transactions on Aerospace and Electronic Systems*, September, 1988.

[31] Eshelman, L., Ehret, D., McDermott, J, and Tan, M., "MOLE: A tenacious knowledge acquisition tool," *Proceedings of the Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada, 1986.

[32] Etherington, D.W. "Formalizing Nonmonotonic Reasoning Systems," *Artificial Intelligence* 31 (1987), pp. 41–85.

[33] Fikes, R.E., "A Commitment-based Framework for Describing Informal Cooperative Work," *Cognitive Science*, 6, pp. 331–347, 1982.

[34] Fikes, R., Morris, P. and Nado, B., "Use of Truth Maintenance in Automatic Planning," *DARPA Knowledge-based Planning Workshop*, Austin, TX, 1987.

[35] Fikes, R.E. and Henderson, D.A., "On Supporting the Use of Procedures in Office Work," *Proceedings of the AAAI-80*, 1980.

[36] Genesereth, M.R. and Nilsson, N.J. *Logical Foundations of Artificial Intelligence*, Palo Alto, CA: Morgan Kaufmann, 1987.

[37] Georgeff, M.P., "Reasoning about Plans and Actions," *Exploring Artificial Intelligence*, H. Shrobe (editor), pp. 173–196, Morgan Kaufmann, 1988.

[38] Ginsberg, A., Weiss, S. and Politakis, P., "SEEK2: A Generalized Approach to Automatic Knowledge Base Refinement," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 367–374, 1985.

[39] Grosz B. and Sidner, C., "Distributed Know-How and Acting: Research on Collaborative Planning," *Proceedings of the DARPA DAI Workshop*, 1988.

[40] Gruber, T., "Acquiring Strategic Knolwedge from Experts" *Proceedings of the Knowledge Acquisition for Knowledge Based Systems Workshop*, pages 10.0–10.18, American Association for Artificial Intelligence, 1987.

[41] Hayes, P.J., "A Representation for Robot Plans," *Proceedings IJCAI-75*, 181-188, 1975.

[42] Hayes-Roth, Barbara, "A Blackboard Architecture for Control," *Artificial Intelligence*, 26, 1985, 251–321.

[43] Hollnagel, E., "Action Not as Planned: The Phenotype and Genotype of Erroneous Actions," draft, Computer Resources International, Copenhagen, Denmark, 1987.

[44] Huff, K.E. *Plan-based Intelligent Assistance: An Approach to Supporting the Software Development Process*, Ph.D dissertation, University of Massachusetts, September 1989.

[45] Huff, K.E., Lesser V.R. "A Plan-based Intelligent Assistant That Supports the Software Development Process" in *Proceedings of the Third ACM Symposium on Software Development Environments*, Boston, November, 1988.

[46] Huff, K.E., Lesser V.R. *Plan Recognition in Open Worlds*, COINS Technical Report 88–18, University of Massachusetts, Amherst, MA., December 1988.

[47] Huff, K.E., Lesser V.R. *The GRAPPLE Plan Formalism*, COINS Technical Report 87–08, University of Massachusetts, Amherst, MA., 1987.

[48] Kahn, G., Nowlan, S. and McDermott, J., "MORE: An Intelligent Knowledge Acquisition Tool," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 581–584, 1985.

[49] Kautz, H. and Allen, J., "Generalized Plan Recognition," *Proceedings AAAI-86*, Palo Alto, CA: Morgan Kaufmann, pp. 32–37, 1986.

[50] Klaus, G. and Buhr, M., "Philosophishes Woerterbuch," VEB Deutscher Verlag der Wissenschaften, Ost-Berlin, GDR, 1972.

[51] de Kleer, J., "Problem Solving with the ATMS" *Artificial Intelligence* Vol. 28, pp. 197–224, 1986.

[52] Lefkowitz, L.S., "Knowledge Acquisition through Anticipation of Modifications," *PhD thesis*, University of Massachusetts, Amherst, MA, 1987.

[53] Lefkowitz, L.S. and Croft, W.B., "Planning and Execution of Tasks in Cooperative Work Environments," *Proceedings of the 5th IEEE Conference on Artificial Intelligence Applications*, 1989.

[54] Charniak, E., Riesbeck, C. and McDermott, D., "Artificial Intelligence Programming," Lawrence Erlbaum, Hillsdale, 1980.

[55] Mahling, D.E. and Croft, W.B., "An Interface for the Specification of Office Activities," *Proceedings of the IFIP Conference on Office Information Systems*, Linz, Austria, August 1988.

[56] Mahling, D.E. and Croft, W.B., "Relating Human Knowledge of Tasks to the Requirements of Plan Libraries," Technical Report 88-33, University of Massachusett at Amherst, 1988.

[57] Malone, T.W., "What is Coordination Theory?," presented at the *NSF Coordination Theory Workshop*, 1988.

[58] Marcus, S.J., McDermott, J. and Wang, T., "Knowledge Acquistion for Constructive Systems," *Proceedings of the Ninth International Conference on Artificial Intelligence*, Los Angeles, California, 1985.

[59] Morris, P.H. and Nado, R.A., "Representing Actions with an Assumption-based Truth Maintenance System," *Proceedings of the AAAI-86*, 1986.

[60] Morris, P. "Curing Anomalous Extensions." *Proceedings AAAI-87*, Palo Alto, CA: Morgan Kaufmann, 1987, pp. 437–442.

[61] Murray, T., Schultz, K., Clement, J., & Brown, D. (in press), "Dealing with science misconceptions using an analogy based computer program: In Soloway, E. (Ed.)," *Interactive learning environments*, NJ: Ablex Publishing.

[62] Musen, M., Fagan, L., Coombs, D. and Shortliffe, E., "Using a Domain Model to Drive an Interactive Knowledge Editing Tool," *Proceedings of the Knowledge Acquisition for Knowledge Based Systems Workshop*, pp. 33.0–33.11, American Association for Artificial Intelligence, 1986.

[63] Newell, A. and Simon, H., "Human Problem Solving," Prentice-Hall, 1972.

[64] Pea, R.D. and Kurland, D.M., "On the Cognitive Prerequisites of Computer Programming," Technical Report TR 18, Bank Street College New York, 1983.

[65] Pednault, E.P. "Formulating Multiagent, Dynamic-World Problems in the Classical Planning Framework," *Proceedings of the 1986 Workshop on Reasoning About Actions and Plans*. Timberline, Oregon, pp. 47-82.

[66] Polson, P.G. and Kieras, D.E., "A Quantitative Model of the Learning and Performance of Text Editing Knowledge," In L. Borman and B. Curtis, editors, *Human Factors in Computing; CHI '85*, ACM, Inc., New York, 1985.

[67] Rasmussen, J. "What Can Be Learned from Human Error Reports?" In K. Duncan, M. Gruneberg, and D. Wallis (Eds.), Changes in Working Life. John Wiley: London. 1980.

[68] Reason, J., Mycielska, K. Absent-Minded? The Psychology of Mental Lapses and Everyday Errors. Prentice-Hall, Inc., 1982.

[69] Reiter, R. "A Logic for Default Reasoning." *Artificial Intelligence*, 13 (1980), pp. 81–132.

[70] Sacerdoti, E.D. *A Structure for Plans and Behavior*, Elsevier North-Holland, Inc., New York, NY, 1977.

[71] Sathi, A., Morton, T.E. and Roth, S.F., "Callisto: An Intelligent Project Management System," *AI Magazine* Vol. 7, No. 5., 1986.

[72] Schank, R.C., *Dynamic Memory*, Cambridge University Press, 1982.

[73] Schank, R.C. and Abelson, R.P., "Scripts, Plans, Goals and Understanding," Lawrence Erlbaum, 1977.

[74] Schmidt, C.F., "Understanding Human Actions," *Conference on Theoretical Issues in Natural Language Processing*, 1975.

[75] Servan-Schreiber,D., "Artificial Intelligence in Psychiatry," *Journal of Nervous and Mental Disaease*, 174, pp. 191–202, 1983.

[76] Stallman, R.M., and Sussman, G.J., "Forward Reasoning and Dependency-directed Backtracking in a System for Computer-aided Circuit Analysis," *Artificial Intelligence*, 9 (1977), pp. 135–196.

[77] Suchman, L.A., "Office Procedures as Practical Action," *ACM Transaction on Office Information Systems*, 4:320–328, 1984.

[78] Suchman, L.A., "Plans and Situated Actions: The Problem of Human-machine Communiation," Technical Report ISL-6, Xerox Corporation, 1985.

[79] Suthers, D. & Rissland, E., "Ex. Gen: a constraint satisfying example generator," *Computer and Information Science Department Technical Report #88–71*, University of Massachusetts, Amherst, MA., 1988.

[80] Swartout, W. (editor), "DARPA Santa Cruz Workshop on Planning," *AI Magazine* Vol. 9, No. 2., 1988.

[81] Tate, A. "Generating Project Networks," *Proceedings IJCAI-77*, Boston, 888-893, 1977.

[82] Tversky, A., "Features of Similarity," *Psychological Review*, 84(4):327–352, 1977.

[83] Wilkins, D.E., "Practical Planning: Extending the Classical AI Planning Paradigm," Morgan-Kauffman Publishers, San Mateo, CA. 1988.

[84] Wilkins, D.E., "Recovering from Execution Errors in SIPE," *SRI Technical Report 346*, 1985.

[85] Wilkins, D.E. "Domain-Independent Planning; Representation and Plan Generation." Artificial Intelligence, 22 (1984), 269-301.

[86] Woods, W., "Transition network grammars for natural language analysis," *Communications of the ACM*, Vol 13:10, 591–606, 1970.

[87] Woolf, B., &Cunningham, P., "Multiple knowledge sources in intelligent tutoring systems," *IEEE Expert*, Summer, 1987.

[88] Woolf, B., Suthers, D., &Murray, T., "Discourse control for tutoring: Case studies in example generation. To be published as a Computer and Information Science Department Tech Report," University of Massachusetts, Amherst, MA.

[89] Woolf, B. & Murray, T., "A framework for representing tutorial discourse," *International joint conference in artificial intelligence (IJCAI-87)*, Los Altos, CA: Morgan Kaufmann, 1987.

[90] Wright, M. and Fox, M.S., "SRL 1.5 User Manual," Intelligent Systems Laboratory, Carnegie-Mellon University Robotics Institute, 1983.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.